

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！



Go语言程序设计

王 鹏 编著

清华大学出版社

21世纪高等学校规划教材 | 计算机科学与技术

Go语言程序设计

王 鹏 编著

清华大学出版社

北京

内 容 简 介

本书是 Go 语言程序基础教程,其特点是从最基本的语法讲起,并结合 Go 标准库列举了大量实例。即使无任何 Go 基础的读者,通过本书也可以很容易地掌握这门程序设计语言。主要内容包括数据类型、控制结构、数组切片和字典、函数、结构体和方法、接口、并发程序设计、网络编程等。

本书内容新颖、体系合理、逻辑性强,是学习 Go 语言的理想教材。本书几乎所有语法点和知识点都配备有实例,并在每章最后有综合应用举例,全部例子都有源代码并调试通过。凡具有初级计算机知识的读者都能读懂本书。本书可作为高等学校计算机、网络、信息类专业的基础教材,对从事计算机应用和开发的技术人员 also 具有很高的参考价值。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

Go 语言程序设计/王鹏编著.--北京:清华大学出版社,2014

21 世纪高等学校规划教材. 计算机科学与技术

ISBN 978-7-302-34723-1

I. ①G… II. ①王… III. ①程序语言—程序设计—高等学校—教材 IV. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 292348 号

责任编辑:郑寅堃 薛 阳

封面设计:傅瑞学

责任校对:白 蕾

责任印制:宋 林

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社 总 机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

课 件 下 载: <http://www.tup.com.cn>, 010-62795954

印 刷 者:北京富博印刷有限公司

装 订 者:北京市密云县京文制本装订厂

经 销:全国新华书店

开 本:185mm×260mm

印 张:23

字 数:560 千字

版 次:2014 年 1 月第 1 版

印 次:2014 年 1 月第 1 次印刷

印 数:1~2000

定 价:39.00 元

产品编号:054436-01

出版说明

随着我国改革开放的进一步深化,高等教育也得到了快速发展,各地高校紧密结合地方经济建设发展需要,科学运用市场调节机制,加大了使用信息科学等现代科学技术提升、改造传统学科专业的投入力度,通过教育改革合理调整和配置了教育资源,优化了传统学科专业,积极为地方经济建设输送人才,为我国经济社会的快速、健康和可持续发展以及高等教育自身的改革发展做出了巨大贡献。但是,高等教育质量还需要进一步提高以适应经济社会发展的需要,不少高校的专业设置和结构不尽合理,教师队伍整体素质亟待提高,人才培养模式、教学内容和教学方法需要进一步转变,学生的实践能力和创新精神亟待加强。

教育部一直十分重视高等教育质量工作。2007年1月,教育部下发了《关于实施高等学校本科教学质量与教学改革工程的意见》,计划实施“高等学校本科教学质量与教学改革工程”(简称“质量工程”),通过专业结构调整、课程教材建设、实践教学改革、教学团队建设等多项内容,进一步深化高等学校教学改革,提高人才培养的能力和水平,更好地满足经济社会发展对高素质人才的需要。在贯彻和落实教育部“质量工程”的过程中,各地高校发挥师资力量强、办学经验丰富、教学资源充裕等优势,对其特色专业及特色课程(群)加以规划、整理和总结,更新教学内容、改革课程体系,建设了一大批内容新、体系新、方法新、手段新的特色课程。在此基础上,经教育部相关教学指导委员会专家的指导和建议,清华大学出版社在多个领域精选各高校的特色课程,分别规划出版系列教材,以配合“质量工程”的实施,满足各高校教学质量和教学改革的需要。

为了深入贯彻落实教育部《关于加强高等学校本科教学工作,提高教学质量的若干意见》精神,紧密配合教育部已经启动的“高等学校教学质量与教学改革工程精品课程建设工作”,在有关专家、教授的倡议和有关部门的大力支持下,我们组织并成立了“清华大学出版社教材编审委员会”(以下简称“编委会”),旨在配合教育部制定精品课程教材的出版规划,讨论并实施精品课程教材的编写与出版工作。“编委会”成员皆来自全国各类高等学校教学与科研第一线的骨干教师,其中许多教师为各校相关院、系主管教学的院长或系主任。

按照教育部的要求,“编委会”一致认为,精品课程的建设工作从开始就要坚持高标准、严要求,处于一个比较高的起点上。精品课程教材应该能够反映各高校教学改革与课程建设的需要,要有特色风格、有创新性(新体系、新内容、新手段、新思路,教材的内容体系有较高的科学创新、技术创新和理念创新的含量)、先进性(对原有的学科体系有实质性的改革和发展,顺应并符合21世纪教学发展的规律,代表并引领课程发展的趋势和方向)、示范性(教材所体现的课程体系具有较广泛的辐射性和示范性)和一定的前瞻性。教材由个人申报或各校推荐(通过所在高校的“编委会”成员推荐),经“编委会”认真评审,最后由清华大学出版

社审定出版。

目前,针对计算机类和电子信息类相关专业成立了两个“编委会”,即“清华大学出版社计算机教材编审委员会”和“清华大学出版社电子信息教材编审委员会”。推出的特色精品教材包括:

(1) 21 世纪高等学校规划教材·计算机应用——高等学校各类专业,特别是非计算机专业的计算机应用类教材。

(2) 21 世纪高等学校规划教材·计算机科学与技术——高等学校计算机相关专业的教材。

(3) 21 世纪高等学校规划教材·电子信息——高等学校电子信息相关专业的教材。

(4) 21 世纪高等学校规划教材·软件工程——高等学校软件工程相关专业的教材。

(5) 21 世纪高等学校规划教材·信息管理与信息系统。

(6) 21 世纪高等学校规划教材·财经管理与应用。

(7) 21 世纪高等学校规划教材·电子商务。

(8) 21 世纪高等学校规划教材·物联网。

清华大学出版社经过三十多年的努力,在教材尤其是计算机和电子信息类专业教材出版方面树立了权威品牌,为我国的高等教育事业做出了重要贡献。清华版教材形成了技术准确、内容严谨的独特风格,这种风格将延续并反映在特色精品教材的建设中。

清华大学出版社教材编审委员会

联系人:魏江江

E-mail: weijj@tup.tsinghua.edu.cn

前言

Go 语言是 Google 推出的一门新的程序开发工具,它具有 C/C++ 的高效性和 Python 的优雅性,是开发 Server 端程序的利器,Google 完全支持 Go,其后台服务器大部分基于 Go 开发,它被称为“未来互联网上的 C 语言”。

作为一门新近推出的静态语言,要被软件开发者所承认、接受,Go 必须有优于其他静态语言的一些特点。随着计算机技术和计算机网络的迅猛发展,软件开发者对程序设计语言提出了新的要求,比如:支持高并发、支持多核心架构,同时开发人员还要求编译速度够快、执行效率更高。作为传统成熟的语言,如 C/C++、Java、.NET、Python 等,它们可能会具备其中一种特点,但不能兼顾。而 Go 语言恰恰是在“快速编译、高效执行、易于开发”这三个条件之间做到了最佳平衡的一种程序设计语言。

另外,Go 语言支持并发,它使用 Goroutine 和 Channel 实现更轻量级的并发,提高了系统实现并行计算的效率,优于系统提供的进程和线程,所以它也是未来云计算的利器。

最后,Go 语言很容易上手,只要具有 C/C++ 或 Java 的基础,它支持 UNIX/Linux、Windows、Mac 等主流平台,可以通过交叉编译很轻松地实现跨平台。

本书详细介绍了 Go 语言的发展历程、特性及程序设计思想。全书共有 11 章,第 1 章介绍了 Go 的版本、下载和安装,以及常用命令。还介绍了集成开发工具 Sublime Text 2。第 2 章介绍了 Go 基本数据类型、运算符和表达式。第 3 章介绍了顺序结构程序设计方法,介绍了 fmt 包和标准输入输出,还通过 strings 包和 strconv 包介绍了字符串处理方法。第 4 章介绍了选择结构程序设计方法,主要包括 if 语句和 switch 语句。第 5 章介绍了循环结构程序设计方法,以及三种跳转语句。第 6 章介绍了构造类型和引用类型,重点介绍了数组切片 Slice,并通过 bytes 包详细说明了大量的 Slice 处理函数。第 7 章介绍了函数,包括 Go 函数的新特性,比如像多返回值、变参、匿名函数、闭包等。第 8 章介绍了结构体和方法,Go 其实是使用结构体来实现面向对象编程的,所以这一章涵盖了大量的 Go 面向对象程序设计知识。第 9 章介绍了接口,在 Go 语言中,接口是用来配合结构体实现面向对象编程的。本章还通过综合实例“二叉树”,完整诠释了 Go 面向对象编程思想及方法。第 10 章介绍了并发程序设计,从程序层面就支持并发设计是 Go 最鲜明的特性,也是它的最大的优势。第 11 章介绍了网络编程,Go 是并发的、面向网络的,所以网络编程是它最基本的功能,也是它展示强大功能的舞台。

本书是作者一直跟踪 Go 语言发展历程的知识积累和经验总结,Go 语言是一个非常年轻的程序设计语言,说它是新生儿也不为过。Google 2009 年才发布了 Go 的 1.03 测试版,2013 年 5 月才发布了它的第一个正式版 1.1 版。所以,本教材中的基础知识大部分来自于 Go 官方文档,另一部分知识来自于 Go 爱好者之间的交流。当然,作者也付出了艰辛和努力,对这些零散的知识进行归纳总结、系统化,并编写了大量的实例代码。

在阅读本书的时候,读者会深深地体会到本书的特点,那就是细致入微地诠释了 Go 语言的每一个知识点,并为每一个知识点编写了实例。这样做的原因,一是 Go 语言非常年轻,几乎没有中文版的教程;二是读者在阅读的时候,可能找不到为你解惑的老师。所以对于有深厚语言功底的读者,可以选择性地阅读这些例子。

本书可作为高等学校计算机、网络、信息类专业的基础教材,对从事计算机应用和开发的技术人员 also 具有很高的参考价值,也可以作为广大程序爱好者自学用书。

本书由陕西理工学院网络工程教研室王鹏老师编著。在本书撰写过程中,得到了 Mark Summerfield 博士和 Go Web 编程交流群众多好友的热情支持与指导,在此一并表示衷心感谢。

由于作者水平有限,加之时间仓促,书中疏漏和不当之处在所难免,敬请读者批评指正。

编者

2013 年 8 月

目 录

第 1 章 Go 语言概述	1
1.1 Go 语言简介	1
1.1.1 Go 语言发展历史	1
1.1.2 Go 语言的特点	1
1.2 Go 的下载和安装	3
1.2.1 源码安装	3
1.2.2 标准包安装	4
1.2.3 第三方工具安装	10
1.3 Go 基本命令及使用	11
1.3.1 Go 常用命令	11
1.3.2 Go 文档查看命令	12
1.3.3 其他命令	12
1.4 Go 集成开发工具	13
1.4.1 LiteIDE	13
1.4.2 Sublime Text 2	14
1.5 Go 程序结构和设计过程	20
1.5.1 Go 程序结构	20
1.5.2 Go 程序设计过程	22
1.5.3 Go 源程序语法要点	23
1.5.4 Go 的注释方式	23
小结	24
习题	25
第 2 章 Go 数据类型、运算符与表达式	26
2.1 常量、变量与命名规则	26
2.1.1 常量	26
2.1.2 变量	27
2.1.3 标识符与命名规则	28
2.2 基本数据类型	29
2.2.1 布尔型数据	29
2.2.2 整型数据	30
2.2.3 浮点型数据	32

2.2.4	复数	34
2.2.5	字节型数据	35
2.2.6	rune 类型	36
2.2.7	uintptr 类型	37
2.3	运算符与表达式	39
2.3.1	赋值运算符	39
2.3.2	算术运算符	40
2.3.3	关系运算符	41
2.3.4	逻辑运算符	42
2.3.5	位运算符	43
2.3.6	通道运算符	44
2.3.7	运算符的优先级和结合性	44
2.4	字符串	45
2.4.1	字符串定义	45
2.4.2	字符串操作	46
2.4.3	字符串遍历	47
2.5	常量的初始化规则	48
2.5.1	常量的类型	48
2.5.2	常量定义方法	49
2.5.3	常量的初始化规则	50
2.6	枚举	53
2.6.1	枚举类型的定义	53
2.6.2	iota 使用规则	54
2.6.3	iota 应用举例	54
2.7	变量的定义与声明	56
2.7.1	变量的类型	56
2.7.2	变量的类型零值	56
2.7.3	变量的作用域	58
2.7.4	变量的声明与赋值	59
2.8	类型别名	62
2.8.1	类型别名定义方式	62
2.8.2	中文类型名	62
2.9	类型转换	63
2.9.1	类型转换方法	63
2.9.2	类型兼容性	64
2.9.3	类型转换分类	64
小结	67
习题	68

第 3 章 Go 顺序结构程序设计	70
3.1 顺序结构程序设计和基本语句	70
3.1.1 顺序程序结构	70
3.1.2 简单语句	70
3.1.3 复合语句	72
3.2 Go 程序语法注意事项	73
3.2.1 Go 程序语句和分号的使用	74
3.2.2 Go 程序语句块和左大括号约定	74
3.2.3 注释语句	74
3.3 数据输入输出	74
3.3.1 标准输出函数	75
3.3.2 标准输入函数	82
3.4 Strings 包	85
3.4.1 字符串查找函数	85
3.4.2 字符串比较函数	87
3.4.3 字符串位置索引函数	87
3.4.4 字符串追加和替换函数	90
3.5 Strconv 包	91
3.5.1 数值转换为字符串函数	91
3.5.2 字符串转换为数值函数	93
3.5.3 Atoi() 和 Itoa() 函数	94
3.6 顺序结构程序举例	95
3.6.1 求平均值	95
3.6.2 计算三角形面积周长	96
3.6.3 求解一元二次方程	97
小结	98
习题	98
第 4 章 Go 选择结构程序设计	100
4.1 if 语句	100
4.1.1 if 语句的形式	100
4.1.2 if 语句的嵌套	104
4.1.3 if 语句的注意事项	105
4.2 switch 语句	107
4.2.1 switch 语句结构	107
4.2.2 switch 语句的特殊形式	108
4.2.3 switch 语句的注意事项	112
4.3 选择结构程序举例	112

4.3.1	解一元二次方程	112
4.3.2	打印中文日期信息	114
	小结	115
	习题	115
第5章 Go 循环结构程序设计		116
5.1	for 语句	116
5.1.1	for 基本循环结构	116
5.1.2	for 条件循环结构	118
5.1.3	for 无限循环结构	119
5.1.4	使用 for 语句的注意事项	121
5.1.5	for 循环嵌套结构	122
5.2	跳转语句	122
5.2.1	break 语句	123
5.2.2	continue 语句	124
5.2.3	goto 语句	126
5.3	for range 语句	128
5.4	循环控制程序举例	129
5.4.1	多重循环嵌套应用举例	129
5.4.2	无限循环和跳转语句应用举例	130
5.4.3	for range 语句应用举例	132
	小结	133
	习题	134
第6章 数组、切片和字典		135
6.1	数组	135
6.1.1	数组的声明	135
6.1.2	数组的初始化	136
6.1.3	数组元素的访问和遍历	137
6.1.4	多维数组	139
6.2	切片	140
6.2.1	切片的声明与创建	141
6.2.2	切片元素的访问和遍历	144
6.2.3	切片的操作	144
6.3	字典	146
6.3.1	字典的声明	146
6.3.2	字典的初始化和创建	147
6.3.3	字典的访问和操作	148
6.4	Go 语言内存分配机制	150

6.4.1	new 函数	150
6.4.2	make 函数	151
6.5	字节切片标准库	152
6.5.1	字节切片处理函数	152
6.5.2	Buffer 创建函数及操作方法	168
6.5.3	Reader 对象及方法	176
6.6	程序举例	181
6.6.1	数组应用	181
6.6.2	Slice 应用	182
6.6.3	Map 应用	183
	小结	184
	习题	185
第 7 章	函数	186
7.1	函数声明	186
7.1.1	函数声明基本格式	186
7.1.2	函数声明举例	187
7.2	函数调用	188
7.2.1	调用标准函数	189
7.2.2	调用自定义函数	189
7.2.3	调用外部包中的函数	191
7.2.4	调用内置函数	192
7.3	参数传递和返回值	192
7.3.1	参数传递	193
7.3.2	返回值	196
7.4	变参函数	198
7.4.1	变参函数的声明	198
7.4.2	变参的传递	199
7.4.3	任意类型的变参	200
7.5	匿名函数与闭包	201
7.5.1	匿名函数	201
7.5.2	闭包	203
7.6	函数的递归调用和 defer 语句	204
7.6.1	函数的递归调用	204
7.6.2	defer 语句	205
7.6.3	异常恢复机制	208
7.7	程序举例	210
7.7.1	函数嵌套调用举例	210
7.7.2	变参函数举例	211

7.7.3	多返回值函数举例	212
	小结	213
	习题	214
第8章	结构体和方法	215
8.1	结构体的定义	215
8.1.1	结构体定义	215
8.1.2	结构体变量	217
8.1.3	结构体对象	218
8.1.4	结构体对象初始化	220
8.1.5	结构体的赋值和关系操作	221
8.2	嵌入式结构	222
8.2.1	嵌入式结构用作字段	222
8.2.2	嵌入式结构直接定义结构体变量	223
8.2.3	嵌入式结构直接用于 Map	224
8.3	匿名字段	225
8.3.1	匿名字段的初始化	226
8.3.2	匿名字段的访问	227
8.3.3	匿名字段的多种形式	228
8.3.4	匿名字段的重名	229
8.3.5	匿名类型指针	231
8.4	方法	231
8.4.1	结构化程序设计思想	231
8.4.2	面向对象程序设计思想	232
8.4.3	Method 的基本定义	233
8.4.4	多个 Method 可以同名	234
8.4.5	指针作为 Receiver	235
8.4.6	匿名 Receiver	237
8.4.7	Method 的继承	237
8.4.8	Method 的重写	238
8.5	可见性规则和 Struct 的导入	239
8.5.1	可见性规则	240
8.5.2	Struct 的导入	240
8.6	字段标签	242
8.7	数据 I/O 对象及操作	243
8.7.1	ReadWriter 对象	243
8.7.2	Reader 对象	243
8.7.3	Writer 对象	249
8.8	应用举例——链表操作	253

8.8.1	链表简介	253
8.8.2	Struct 和 Method 设计单链表	254
小结		257
习题		257
第 9 章	接口	259
9.1	接口的概念与定义	259
9.1.1	接口的概念	259
9.1.2	接口的定义	259
9.1.3	接口组合	260
9.1.4	空接口	261
9.2	接口执行机制和赋值	261
9.2.1	接口执行机制	261
9.2.2	接口的赋值	262
9.3	匿名字段方法和接口转换	263
9.3.1	匿名字段方法	264
9.3.2	接口转换	265
9.4	接口类型推断	266
9.4.1	Comma-ok 断言	266
9.4.2	Switch 测试	267
9.5	反射	268
9.5.1	获取原对象的 Type 和 Value 值	268
9.5.2	修改原对象 Value 值	272
9.5.3	动态调用原对象方法	273
9.6	应用举例——二叉树	274
9.6.1	树的定义和基本术语	275
9.6.2	二叉树简介	275
9.6.3	二叉树的链接存储结构	276
9.6.4	二叉树基本应用测试	281
小结		284
习题		284
第 10 章	Go 并发程序设计	286
10.1	程序并发执行概述	286
10.1.1	程序的顺序执行	286
10.1.2	程序的并发执行	287
10.1.3	程序的并行执行	287
10.2	Goroutine	288
10.2.1	操作系统提供的并发基础	288

10.2.2	Goroutine 的定义	288
10.2.3	Goroutine 的创建	289
10.3	Channel	290
10.3.1	程序间的并发通信	290
10.3.2	Channel 简介	290
10.3.3	Channel 声明和初始化	291
10.3.4	数据接收和发送	291
10.3.5	Channel 的关闭和迭代器	293
10.3.6	单向 Channel	294
10.3.7	异步 Channel	295
10.4	Select 机制和超时机制	297
10.4.1	Select 机制	297
10.4.2	超时机制	299
10.5	Runtime Goroutine	300
10.5.1	出让时间片	300
10.5.2	获取 CPU 核心数和任务数	301
10.5.3	终止当前 Goroutine	302
	小结	303
	习题	303
第 11 章	Go 网络编程	305
11.1	Go 网络编程简介	305
11.1.1	计算机网络概念和体系结构	305
11.1.2	网络编程基本概念	307
11.1.3	网络编程模式	311
11.1.4	Socket 网络编程接口	313
11.2	Go 网络编程基础	314
11.2.1	IP 地址和域名解析	314
11.2.2	主机信息查询	318
11.2.3	服务信息查询	320
11.3	Go 网络编程原理	322
11.3.1	Socket 网络编程	323
11.3.2	Go 网络编程	323
11.4	TCP 网络程序设计	324
11.4.1	TCPAddr 地址结构体	324
11.4.2	TCPConn 对象	325
11.4.3	TCP 服务器设计	326
11.4.4	TCP 客户机设计	328
11.4.5	使用 Goroutine 实现并发服务器	329

11.5	UDP 网络程序设计	331
11.5.1	UDPAddr 地址结构体	331
11.5.2	UDPConn 对象	331
11.5.3	UDP 服务器设计	332
11.5.4	UDP 客户机设计	333
11.6	IP 网络程序设计	335
11.6.1	IPAddr 地址结构体	335
11.6.2	IPConn 对象	335
11.6.3	IP 服务器设计	336
11.6.4	IP 客户机设计	337
11.6.5	Ping 程序设计	339
	小结	341
	习题	342
附录 A	Go 语言内置关键字	343
附录 B	Go 内置函数	344
附录 C	Go 语言标准库	345
附录 D	名词与术语索引表	346
参考文献	Go 语言发展历史	351

Go 语言作为一个开源项目,起源 2007 年,并于 2007 年 9 月 21 日完成雏形设计。开源(Open Source,开放源码)被非赢利软件组织(美国的 Open Source Initiative 协会)注册为认证标记,并对其进行了正式的定义,用于描述那些源码可以被公众使用的软件,并且此软件的使用、修改和发行也不受许可证的限制。

自 2008 年 1 月起,Ken Thompson 开始研发一款以 C 语言为目标结果的编译器来拓展 Go 语言的设计思想,并于 2008 年 5 月完成了第一个 gcc 前端设计。2009 年 10 月 30 日,Russ Cox 完成了 Go 语言标准包的开发。2009 年 11 月 10 日 Google 正式发布 Go 1.03,Go 语言项目以 BSD-style 授权(完全开源)方式,正式公布了在 Linux 和 Mac OS X 平台上的版本。同年 11 月 22 日,Hector Chu 公布了 Go 语言项目的 Windows 版本。2013 年 5 月 13 日,Google 发布了 Go 1.1 版本。

作为一个开源项目,Go 语言借助开源社区的有生力量得到了迅速发展,并吸引了很多 Go 的爱好者使用并完善它。Go 语言在 2010 年 1 月 8 日被 Tiobe(闻名于它的编程语言流行程度排名)宣布为“2009 年年度语言”,引起各界很大的反响。

1.1.2 Go 语言的特点

作为一门新近推出的静态语言,要被软件开发者所承认、接受,Go 必须有优于其他静态语言的一些特点。随着计算机技术和计算机网络的迅猛发展,软件开发者对程序设计语言提出了新的要求,比如:支持高并发、支持多核心架构,同时开发人员还要求编译速度够快、

第1章

Go语言概述

Go语言是一种开源的编译型程序设计语言,它支持并发、垃圾回收机制以提升应用程序性能。它既具有像C这种静态编译型语言的高性能,又具备像Python这种动态语言的高效性,它被称为“未来互联网上的C语言”。

1.1 Go语言简介

Go语言是由贝尔实验室包括肯·汤普森(Ken Thompson)在内的Plan 9原班团队开发的。Go设计的目标是要应对软件开发所面临的最新挑战,比如:自动垃圾回收、快速编译、支持并发、支持多核心架构等。

1.1.1 Go语言发展历史

Go语言作为一个开源项目,起源2007年,并于2007年9月21日完成雏形设计。开源(Open Source,开放源码)被非赢利软件组织(美国的Open Source Initiative协会)注册为认证标记,并对其进行了正式的定义,用于描述那些源码可以被公众使用的软件,并且此软件的使用、修改和发行也不受许可证的限制。

自2008年1月起,Ken Thompson开始研发一款以C语言为目标结果的编译器来拓展Go语言的设计思想,并于2008年5月完成了第一个gcc前端设计。2009年10月30日,Russ Cox完成了Go语言标准包的开发。2009年11月10日Google正式发布Go 1.03,Go语言项目以BSD-style授权(完全开源)方式,正式公布了在Linux和Mac OS X平台上的版本。同年11月22日,Hector Chu公布了Go语言项目的Windows版本。2013年5月13日,Google发布了Go 1.1版本。

作为一个开源项目,Go语言借助开源社区的有生力量得到了迅速发展,并吸引了很多Go的爱好者使用并完善它。Go语言在2010年1月8日被Tiobe(闻名于它的编程语言流行程度排名)宣布为“2009年年度语言”,引起各界很大的反响。

1.1.2 Go语言的特点

作为一门新近推出的静态语言,要被软件开发者所承认、接受,Go必须有优于其他静态语言的一些特点。随着计算机技术和计算机网络的迅猛发展,软件开发者对程序设计语言提出了新的要求,比如:支持高并发、支持多核心架构,同时开发人员还要求编译速度够快、

执行效率更高。作为传统成熟的语言,如 C/C++、Java、.NET、Python 等,它们可能会具备其中一种特点,但不能兼顾。而 Go 语言恰恰是在“快速编译、高效执行、易于开发”这三个条件之间做到了最佳平衡的一种程序设计语言。

1. 快速编译

Go 语言编译速度快主要有以下几个原因:

首先,Go 不使用头文件,在 C 语言开发中,用户经常要导入一些头文件(include),但是在经过一系列修改之后,开发人员可能不知道是否引用了这些头文件中的内容。在这种情况下编译程序时,一是导致编译速度会比较慢,另外编译了不必要的内容,无谓地增大了程序的体积。

其次,Go 在编译时会自动进行检查,判断程序和包之间是否有必要的联系。如果没有联系,Go 会报告编译异常,会强迫程序不导入多余的包,这样也可以减轻程序体积,加快编译的速度。

最后,Go 的对象模块里面包含足够的依赖关系信息,所以编译器不需要重新创建文件。开发人员只需要简单地编译主模块,项目中需要的其他部分就会自动编译。

2. 并发设计易于实现

相对于其他语言,Go 程序的并行、并发设计更加容易,Go 使用 goroutine 这种轻量级线程来实现并发,然后通过 channel 来实现各个 goroutine 之间的通信。在任意函数前面加上“go”关键字,该函数就会在其 goroutine 线程中自动运行。goroutine 通过 channel 通信时屏蔽了几乎所有消息队列,使得程序并发执行效率很高。

Go 语言使用 goroutine 实现高并发的特性,使得应用程序能更好地利用大量的分布式和多核的计算机,这个特性也是 Go 语言强于其他语言的明证,它不仅支持了日益普及的多核与多处理器计算机,也弥补了现存编程语言在这方面所存在的不足。

3. 高效的垃圾回收机制

使用 C++ 时,内存泄漏是长期困扰设计者们的一个难题,Go 语言改变了这种现状。使用 Go 开发时,设计者们无须关心内存管理问题,Go 已经帮助设计者解决了这个问题。在 Go 中,尽管还是像其他静态语言一样执行本地代码,但实质上,代码可以被看作是在某种意义的虚拟机上执行的。Go 以此来实现高效快速的垃圾回收(使用了一个简单的标记-清除算法)。尽管垃圾回收并不容易实现,但考虑这将是未来并发应用程序发展的一个重要组成部分,Go 语言的设计者们还是完成了这项艰难的任务。

另外,Go 还有一些其他优点,比如:Go 语言是一门类型安全和内存安全的编程语言。虽然 Go 语言中仍有指针的存在,但并不允许进行指针运算;对 UTF-8 编码的支持,每一个 Go 源代码文件都是 UTF-8 格式的,用户可输入任何字符,这是其他语言所不支持的;优秀的错误机制,例如,一个死锁程序,在 Go 运行时会通知开发人员目前哪个线程导致了这种死锁,编译器提供的错误信息非常详细全面。

1.2 Go 的下载和安装

Go 语言支持主流的操作系统平台：UNIX/Linux、Windows、Mac，它使用交叉编译技术很容易就可以实现跨平台。例如，可以在 Windows 系统下编译运行 Linux 下的 Go 程序，而无须做任何修改工作。

Go 是一个类似于 BSD 许可发行的程序开发语言，目前官方提供了两个版本的编译器：gc 编译器和 gccgo 编译器。gcc 编译器都是在类 UNIX 系统下工作，也可以通过安装 MinGW 从而在 Windows 平台下使用 gcc 编译器。gc 编译器可以在 FreeBSD、Linux、Mac OS X 以及 Windows 等操作系统和核心架构上运行。

Go 最常用的安装方法有：源码安装、标准包安装和第三方工具安装，可以选择适合自己的方法进行安装。

1.2.1 源码安装

这是一种标准的软件安装方式。对于经常使用 UNIX 类系统的用户，尤其对于开发者来说，从源码安装是最方便而熟悉的。在 Go 的源代码中，有些部分是用 Plan 9 C 和 AT&T 汇编写的，因此假如要想从源码安装，就必须安装 C 的编译工具。

(1) 在 Mac 系统中，只要安装了 Xcode，就已经包含相应的编译工具。

(2) 在类 UNIX 系统中，需要安装 gcc 等工具。例如，Ubuntu 系统可通过在终端中执行 `sudo apt-get install gcc libc6-dev` 来安装编译工具。

(3) 在 Windows 系统中，需要安装 MinGW，然后通过 MinGW 安装 gcc，并设置相应的环境变量。

Go 使用 Mercurial 进行版本管理，首先必须安装 Mercurial，然后才能下载。安装好 Mercurial 后，到 Go 的安装目录 `$GO_INSTALL_DIR` 下，执行如下代码：

```
hg clone -u release https://code.google.com/p/go
cd go/src
./all.bash
```

运行 `all.bash` 后出现“ALL TESTS PASSED”字样时才算安装成功。

上面是 UNIX 风格的命令，Windows 下的安装方式类似，只不过是运行 `all.bat`，调用的编译器是 MinGW 的 gcc。

最后设置以下几个环境变量：

```
export GOROOT=$HOME/go
export GOBIN=$GOROOT/bin
export PATH=$PATH:$GOBIN
```

如果出现 Go 的 Usage 信息，那么说明 Go 已经安装成功了；如果出现该命令不存在，那么可以检查一下自己的 PATH 环境变量中是否包含 Go 的安装目录。

1.2.2 标准包安装

Go 提供了方便的安装包,支持 Windows、Linux、Mac 等系统。这种方式适合初学者,可根据自己的系统位数下载好相应的安装包,安装过程十分简单,本教程主要介绍这种安装方法。

1. Go 标准包的版本

Google 目前发布了 Go 1.0.3 正式版和 Go 1.1 beta2 测试版,Go 标准包各版本所适用的操作系统平台和核心架构如表 1-1 所示。正式版(Official Version)是在测试版或试用版后正式发布的版本,测试版(Test Version)是软件正式发布前发行的版本。软件的测试版一般有三个阶段:Alpha(α)、Beta(β)和 Gamma(γ)。Alpha 是内测版本,即现在所说的 CB,指的是开发团队内部测试或者供有限用户体验测试的版本。Beta 是公测版本,是对所有用户开放的测试版本。Gamma 版本,现在一般叫做 RC(Release Candidate),指的是软件版本正式发行的候选版。

表 1-1 Go 标准包适用的操作系统和核心架构

安 装 包	OS 平台	核心架构
Go 1.0.3. darwin-386. tar. gz	Mac OS X	x86 32-bit
Go 1.0.3. darwin-amd64. tar. gz	Mac OS X	x86 64-bit
Go 1.0.3. freebsd-amd64. tar. gz	FreeBSD	x86 64-bit
Go 1.0.3. linux-386. tar. gz	Linux	x86 32-bit
Go 1.0.3. linux-amd64. tar. gz	Linux	x86 64-bit
Go 1.0.3. windows-386. msi	Windows	x86 32-bit
Go 1.0.3. windows-amd64. msi	Windows	x86 64-bit

本教程所有示例代码的调试与测试都是在 x86 32-bit 架构、Windows 操作系统平台上进行,所以选取的标准包是 Go 1.0.3. windows-386. msi。

2. Go 标准包的下载

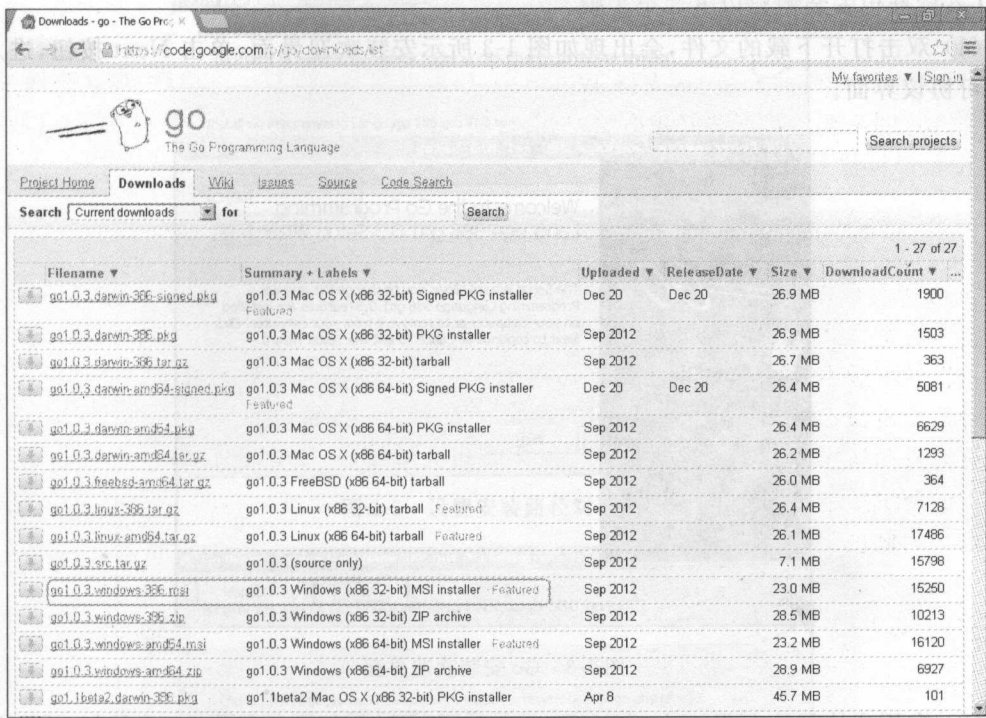
Go 支持开源,所以用户只需使用官网提供的下载连接,就可以很方便地下载安装包进行安装,图 1-1 是 Go 官网标准包下载列表。

Go 标准包官网下载地址: <https://code.google.com/p/go/downloads/list>。

3. 在 Windows 系统中安装 Go 标准包

本教程是在清楚操作系统和核心架构的前提下,下载 Go 1.0.3. windows-386. msi 进行安装演示。其他用户如果在不清楚自己平台特性的情况下,随意下载一个版本进行安装,很可能导致安装失败。所以对于其他用户,在下载安装 Go 之前,首先要判断自己操作系统的位数,然后再选择一个合适的版本。

Windows 系统用户请按 Win+R 键,运行 cmd 程序,输入“systeminfo”命令后回车,稍等片刻,会出现一些系统信息。在“系统类型”一行中,若显示“x64-based PC”,即为 64 位系统;若显示“X86-based PC”,则为 32 位系统,如图 1-2 所示。



Filename	Summary + Labels	Uploaded	ReleaseDate	Size	DownloadCount
go1.0.3.darwin-386-signed.pkg	go1.0.3 Mac OS X (x86 32-bit) Signed PKG installer	Dec 20	Dec 20	26.9 MB	1900
go1.0.3.darwin-386.pkg	go1.0.3 Mac OS X (x86 32-bit) PKG installer	Sep 2012		26.9 MB	1503
go1.0.3.darwin-386.tar.gz	go1.0.3 Mac OS X (x86 32-bit) tarball	Sep 2012		26.7 MB	363
go1.0.3.darwin-amd64-signed.pkg	go1.0.3 Mac OS X (x86 64-bit) Signed PKG installer	Dec 20	Dec 20	26.4 MB	5081
go1.0.3.darwin-amd64.pkg	go1.0.3 Mac OS X (x86 64-bit) PKG installer	Sep 2012		26.4 MB	6629
go1.0.3.darwin-amd64.tar.gz	go1.0.3 Mac OS X (x86 64-bit) tarball	Sep 2012		26.2 MB	1293
go1.0.3.freebsd-amd64.tar.gz	go1.0.3 FreeBSD (x86 64-bit) tarball	Sep 2012		26.0 MB	364
go1.0.3.linux-386.tar.gz	go1.0.3 Linux (x86 32-bit) tarball	Sep 2012		26.4 MB	7128
go1.0.3.linux-amd64.tar.gz	go1.0.3 Linux (x86 64-bit) tarball	Sep 2012		26.1 MB	17486
go1.0.3.src.tar.gz	go1.0.3 (source only)	Sep 2012		7.1 MB	15798
go1.0.3.windows-386.msi	go1.0.3 Windows (x86 32-bit) MSI installer	Sep 2012		23.0 MB	15250
go1.0.3.windows-386.zip	go1.0.3 Windows (x86 32-bit) ZIP archive	Sep 2012		28.5 MB	10213
go1.0.3.windows-amd64.msi	go1.0.3 Windows (x86 64-bit) MSI installer	Sep 2012		23.2 MB	16120
go1.0.3.windows-amd64.zip	go1.0.3 Windows (x86 64-bit) ZIP archive	Sep 2012		28.9 MB	6927
go1.1beta2.darwin-386.pkg	go1.1beta2 Mac OS X (x86 32-bit) PKG installer	Apr 8		45.7 MB	101

图 1-1 Go 标准包下载列表

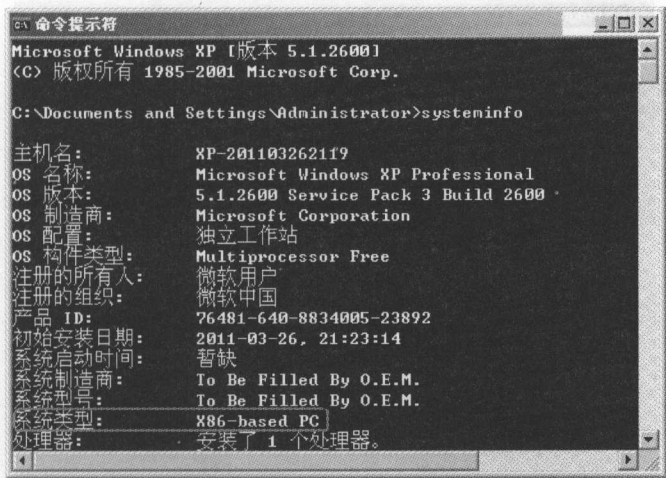


图 1-2 查看系统信息

在 Windows 系统中,Go 标准包会一键安装好,安装路径会默认设置为: c:\Go,如果需要改变安装位置,必须在环境变量中设置如下信息:

```
export GOROOT=$HOME/go
export PATH=$PATH:$GOROOT/bin
```

图 1-7 安装结束

Go 标准包安装需要以下 5 个步骤：

(1) 双击打开下载的文件，会出现如图 1-3 所示安装欢迎界面，单击 Next 按钮，进入用户许可协议界面。

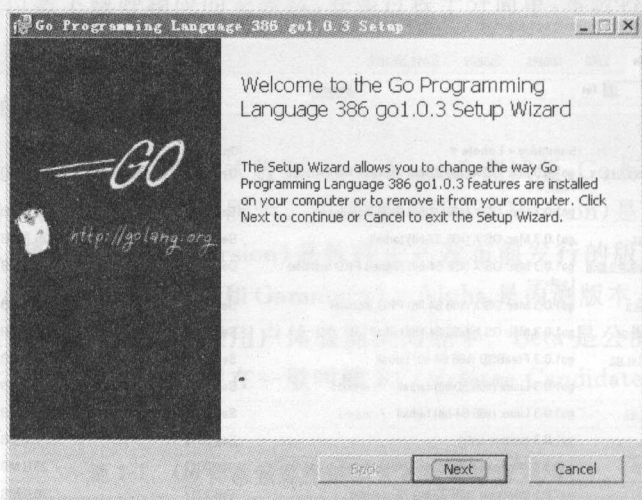


图 1-3 Go 安装欢迎界面

(2) 在用户协议许可界面中，勾选 I accept...复选框(图 1-4)，单击 Next 按钮，进入设置安装路径对话框。

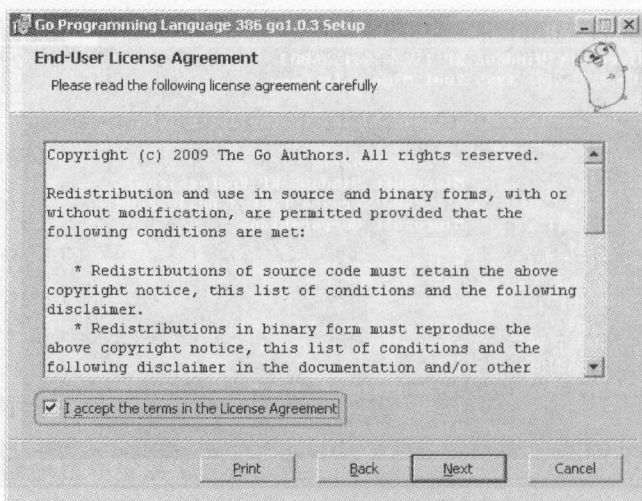


图 1-4 用户许可协议界面

(3) 在设置安装路径对话框中，默认安装路径是：c:\Go，如图 1-5 所示。如果需要更改，在路径输入框中输入新的路径信息，然后单击 Next 按钮；否则，直接单击 Next 按钮，进入准备安装界面。

(4) 进入准备安装界面后，直接单击 Install 按钮(图 1-6)，进入安装过程界面。

(5) 等待安装结束，进入安装结束界面，单击 Finish 按钮(图 1-7)，完成安装。

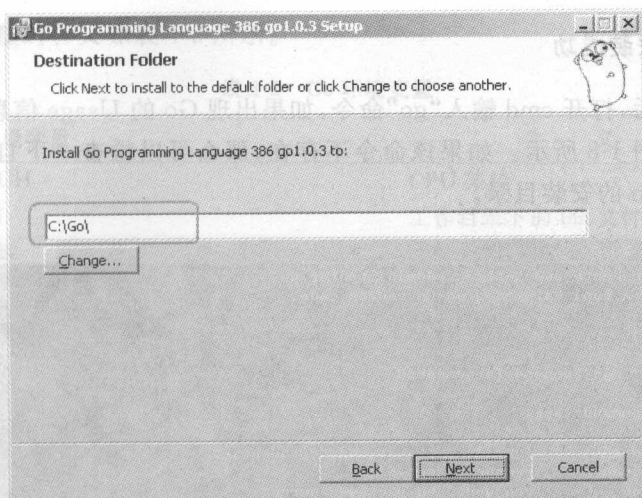


图 1-5 设置安装路径对话框

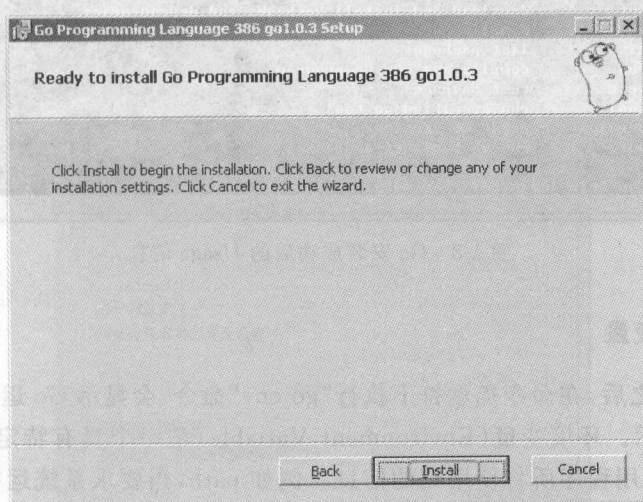


图 1-6 准备安装界面

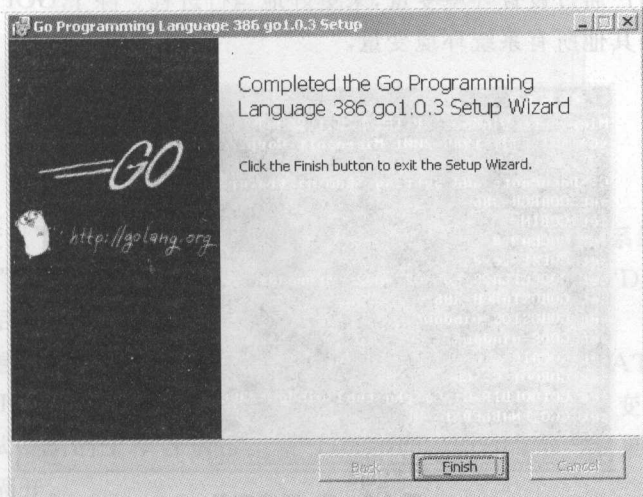


图 1-7 安装结束

4. 检查是否安装成功

Go 安装完成后,打开 cmd 输入“go”命令,如果出现 Go 的 Usage 信息,那么说明 Go 已经安装成功了,如图 1-8 所示。如果该命令不存在,那么可以检查一下自己的 PATH 环境变量中是否包含 Go 的安装目录。

图 1-8 Go 安装成功后的 Usage 信息

5. 配置环境变量

Go 安装成功之后,在命令提示符下执行“go env”命令,会显示 Go 运行所需的系统环境变量,如图 1-9 所示。环境变量(Environment Variable)是一个具有特定名字的对象,它包含一个或者多个应用程序所将使用到的信息。例如 path,当要求系统运行一个程序而没有告诉它程序所在的完整路径时,系统除了在当前目录下面寻找此程序外,还应到 path 中指定的路径去找。用户通过设置环境变量,来更好地运行进程。除了 GOPATH 以外,一键安装包会自动配置好其他所有系统环境变量。

图 1-9 Go 环境变量

Go 各环境变量的含义如表 1-2 所示。

表 1-2 Go 环境变量

环境变量	含 义
GOARCH	CPU 架构
GOBIN	工作目录下的 bin 文件夹
GOEXE	生产可执行文件的后缀
GOHOSTARCH	想要交叉编译的 CPU 架构
GOHOSTOS	想要交叉编译的操作系统
GOOS	当前操作系统
GOPATH	工作目录
GOROOT	安装目录

Go 安装成功后,需在 Windows 系统中手工配置 GOPATH 环境变量,配置步骤如下。

(1) 在“我的电脑”上单击鼠标右键,选择“属性”命令,打开“系统属性”对话框(图 1-10),选择“高级”选项卡,单击“环境变量”按钮,打开“环境变量”设置对话框。

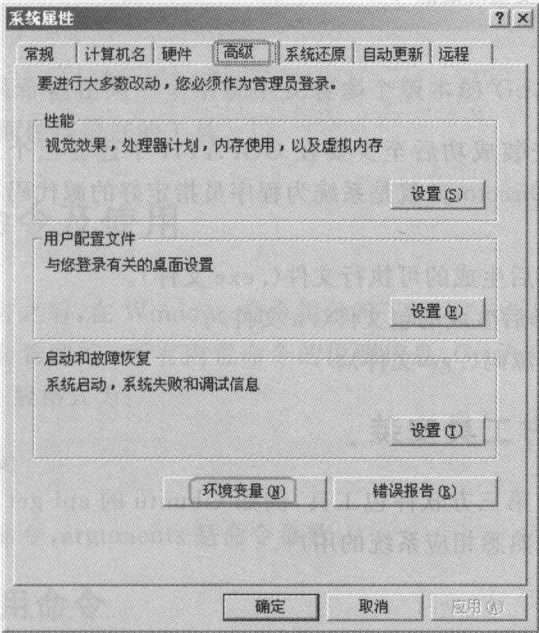


图 1-10 “系统属性”对话框

(2) 在“系统变量”设置区中单击“新建”按钮(图 1-11),打开“新建系统变量”对话框。

(3) 在“变量名”输入框中输入“GOPATH”,在“变量值”输入框中输入“D:\go\development”,如图 1-12 所示,最后单击“确定”按钮完成设置。

GOPATH 设置完成后,在“系统变量”信息显示区域会增加“GOPATH”信息项。需要注意的是,GOPATH 环境变量“变量名”必须是大写的“GOPATH”,“变量值”即为工作目录的路径,本书测试使用的工作目录路径是“D:\go\development”,如果有多个工作目录,路径之间使用“;”隔开。

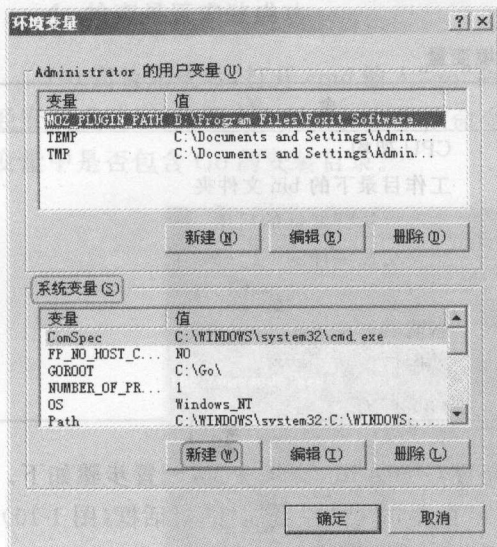


图 1-11 环境变量对话框

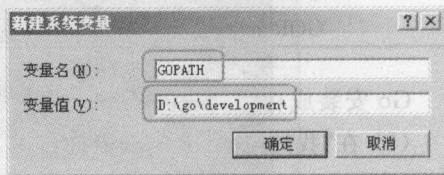


图 1-12 GOPATH 环境变量设置

6. 创建工作目录

Go 语言约定, 在安装成功后至少要在 GOPATH 下建立三个工作目录: bin、pkg 和 src。工作目录(Work Directory)就是系统为程序员指定好的源代码存放、函数调用、数据导入的默认路径。

- (1) bin: 存放编译后生成的可执行文件(.exe 文件)。
- (2) pkg: 存放编译后生成的包文件(.a 文件)。
- (3) src: 存放项目源码(.go 文件)。

1.2.3 第三方工具安装

目前有很多方便的第三方软件包工具, 例如 Ubuntu 的 apt-get、Mac 的 homebrew 等。这种安装方式适合那些熟悉相应系统的用户。

1. GVM

GVM 是第三方开发的 Go 多版本管理工具, 类似 Ruby 里面的 RVM 工具。使用起来相当方便, 安装 GVM 使用如下命令:

```
bash <<(curl -s https://raw.githubusercontent.com/moovweb/gvm/master/binscripts/gvm-installer)
```

GVM 安装完成后, 执行如下两条命令 Go 命令安装就完成了。

```
gvm install go1.0.3
gvm use go1.0.3
```

执行完上面的命令之后 GOPATH、GOROOT 等环境变量会自动设置好,Go 就可以直接使用了。

2. apt-get

Ubuntu 是目前使用最多的 Linux 桌面系统,使用 apt-get 命令来管理软件包,可以通过下面的命令来安装 Go:

```
sudo add - apt - repository ppa:gophers/go
sudo apt - get update
sudo apt - get install golang - stable
```

3. homebrew

homebrew 是 Mac 系统下面目前使用最多的管理软件的工具,目前已支持 Go,可以通过如下命令直接安装 Go:

```
brew install go
```

需要注意的是,如果希望在同一个系统中安装多个版本的 Go,最好使用第三方工具 GVM,这是目前在这方面做得最好的工具。

1.3 Go 基本命令及使用

在成功安装 Go 语言之后,在 Windows 命令提示符下输入“go”,会出现 Go 的 Usage 信息,通过 Usage 信息就会看到 Go 所有内置命令的说明信息,Go 语言带有一套完整的命令操作工具。Go 命令的一般格式为:

```
go command [arguments]
```

其中,command 是操作命令,arguments 是命令参数。

1.3.1 Go 常用命令

Go 常用命令主要有: get、run、build、fmt、install、test 等。

1. go get

go get 动态获取远程代码包,如果是从 GitHub 上远程安装包需提前安装 git,如果是从 Google Code 上远程安装包需提前安装 hg。

2. go run

go run 编译并直接运行程序,它会生成一个临时文件(但不会生成 .exe 文件),直接在命令行输出程序执行结果,方便用户调试。

3. go build

go build 用于测试编译包,主要检查是否会有编译错误,如果是一个可执行文件的源码(即是 main 包),就会直接生成一个可执行文件。

4. go fmt

go fmt 格式化源码,有的 IDE 在保存时就会自动执行这个命令,所以一般不用手动去执行。如果 IDE 不自动执行该命令,可手动执行: go fmt <文件名>.go,就会格式化源代码。go fmt 是 Go 给予的命令,所以可保证所有的 Go 源码格式都是相同的。

5. go install

go install 的作用有两步:第一步是编译导入的包文件,所有导入的包文件编译完才会编译主程序;第二步是将编译后生成的可执行文件放到 bin 目录下(\$GOPATH/bin),编译后的包文件放到 pkg 目录下(\$GOPATH/pkg)。

6. go test

go test 运行测试文件,该命令会自动读取源码目录下面名为 *_test.go 的文件,生成并运行测试用的可执行文件,测试成功会显示“PASS”、“OK”等信息。

1.3.2 Go 文档查看命令

Go 提供 godoc 命令帮助用户查看文档,可以查看函数或者包。例如: godoc fmt.Println 查询 fmt 包中的 println 函数; godoc builtin 查询 Go 内置函数等。

有时用户会觉得在命令行下查看 Go 文档比较麻烦,为此 Go 提供了一个内置命令:

```
go doc - http = :8080
```

该命令可以让用户在本地(localhost)8080 端口,以网页形式查看 Go 帮助文档,就和登录到 Google 网站上查看文档一样,如图 1-13 所示。

1.3.3 其他命令

除了以上那些命令,Go 还提供了如下一些其他命令。

- (1) go clean: 用来移除当前源码包里面编译生成的文件。
- (2) go env: 查看当前 go 的环境变量。
- (3) go fix: 用来修复以前老版本的代码到新版本,例如将 go 1.0 版本的代码更新为 go 1.1 版。
- (4) go list: 列出当前全部安装的 package。
- (5) go version: 查看 go 当前的版本。

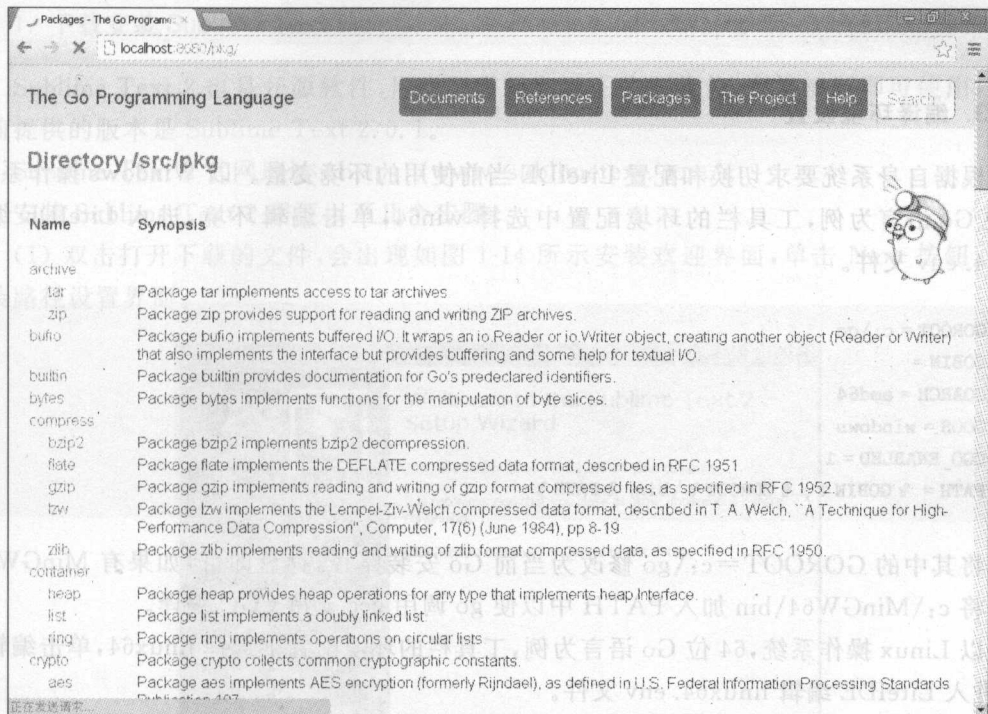


图 1-13 使用 localhost 查看 Go 文档

1.4 Go 集成开发工具

目前 Go 的开发工具主要有: LiteIDE、Sublime Text、Vim、Emacs、Eclipse 等,这些开发工具都各有特点,本教材主要介绍 LiteIDE、Sublime Text 这两种。

1.4.1 LiteIDE

LiteIDE 是一款专门为 Go 语言开发的跨平台轻量级集成开发环境(IDE),由 VisualFC 编写。LiteIDE 支持 Linux、Windows、Mac OS X 等主流操作系统,能通过管理和切换多个 Go 编译环境,同时支持 Go 语言交叉编译。

1. LiteIDE 下载及安装

下载地址: <http://code.google.com/p/golangide>。

源码地址: <https://github.com/visualfc/liteide>。

首先安装好 Go 语言环境,然后根据操作系统下载 LiteIDE 对应的压缩文件直接解压即可使用。

2. 安装 Gocode

使用 go get 命令能自动远程完成 Gocode 的安装,命令如下:

```
go get -u github.com/nsf/gocode
```

3. 编译环境设置

根据自身系统要求切换和配置 LiteIDE 当前使用的环境变量。以 Windows 操作系统, 64 位 Go 语言为例, 工具栏的环境配置中选择 win64, 单击编辑环境, 进入 LiteIDE 编辑 win64.env 文件。

```
GOROOT=c:\go
GOBIN=
GOARCH=amd64
GOOS=windows
CGO_ENABLED=1
PATH=%GOBIN%;%GOROOT%\bin;%PATH%
```

将其中的 GOROOT=c:\go 修改为当前 Go 安装路径, 存盘即可, 如果有 MinGW64, 可以将 c:\MinGW64\bin 加入 PATH 中以便 go 调用 gcc 支持 CGO 编译。

以 Linux 操作系统, 64 位 Go 语言为例, 工具栏的环境配置中选择 linux64, 单击编辑环境, 进入 LiteIDE 编辑 linux64.env 文件。

```
GOROOT=$HOME/go
GOBIN=
GOARCH=amd64
GOOS=linux
CGO_ENABLED=1
PATH=$GOBIN:$GOROOT/bin:$PATH
```

将其中的 GOROOT=\$HOME/go 修改为当前 Go 安装路径, 存盘即可。

4. GOPATH 设置

Go 语言的工具链使用 GOPATH 设置, 是 Go 语言开发的项目路径列表, 在命令行中输入“go help gopath”快速查看 GOPATH 文档(在 LiteIDE 中也可以按 Ctrl+ 键直接输入)。

在 LiteIDE 中可以方便地查看和设置 GOPATH。通过“菜单”→“查看”→“GOPATH 设置”, 可以查看系统中已存在的 GOPATH 列表, 同时可根据需要添加项目目录到自定义 GOPATH 列表中。

1.4.2 Sublime Text 2

Sublime Text 2 的主要特点有: 自动化代码提示功能; 保存时自动格式化代码, 使代码变得更加整齐、美观, 符合 Go 语言标准; 另外, Sublime Text 2 支持项目化管理, 支持语法高亮。在使用 Sublime Text 2 开发 Go 程序时, 通常选择 Sublime+GoSublime+gocode+MarGo 的组合。本教材所有示例代码都使用 Sublime Text 2 编写, 调试运行。

1. 下载安装 Sublime Text 2

Sublime Text 2 也是开源软件,用户只需到官网下载安装包,直接解压即可使用,官网目前提供的版本是 Sublime Text 2.0.1。

Sublime Text 2 官网地址: <http://www.sublimetext.com/>。

安装 Sublime Text 2 需要以下几个步骤。

(1) 双击打开下载的文件,会出现如图 1-14 所示安装欢迎界面,单击 Next 按钮,进入安装路径设置界面。

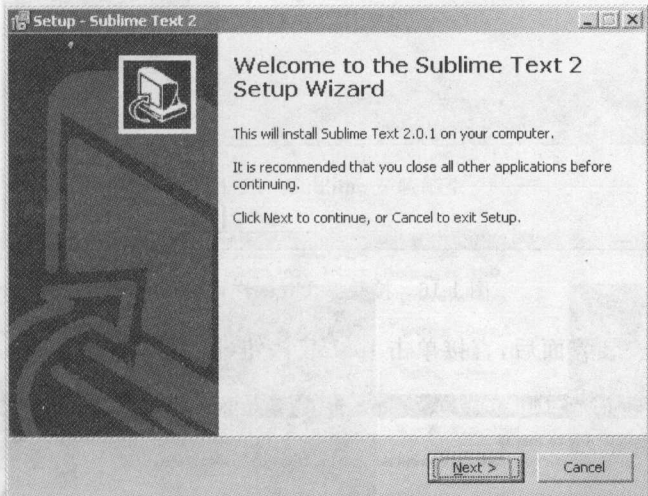


图 1-14 Sublime 安装欢迎界面

(2) 在路径设置对话框中,默认路径是: C:\Program Files\Sublime Text 2,如图 1-15 所示。如果需要更改,在路径输入框中输入新的路径信息,然后单击 Next 按钮;否则,直接单击 Next 按钮,进入用户许可协议界面。

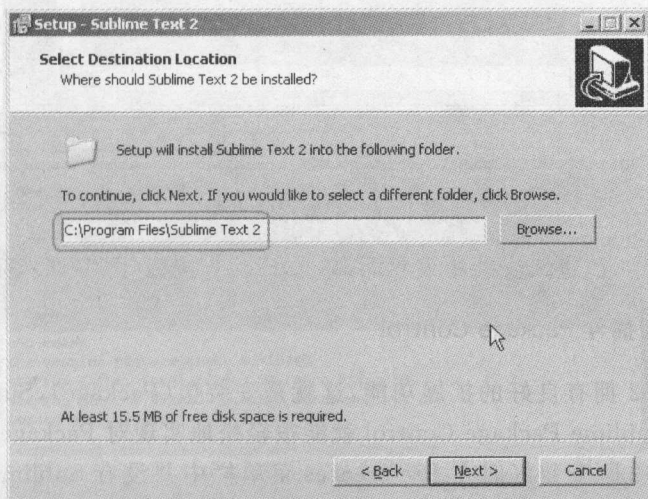


图 1-15 Sublime 安装路径设置界面

(3) 在用户协议许可界面中,勾选 Add to explorer context menu 复选框(图 1-16),单击 Next 按钮,进入准备安装界面。

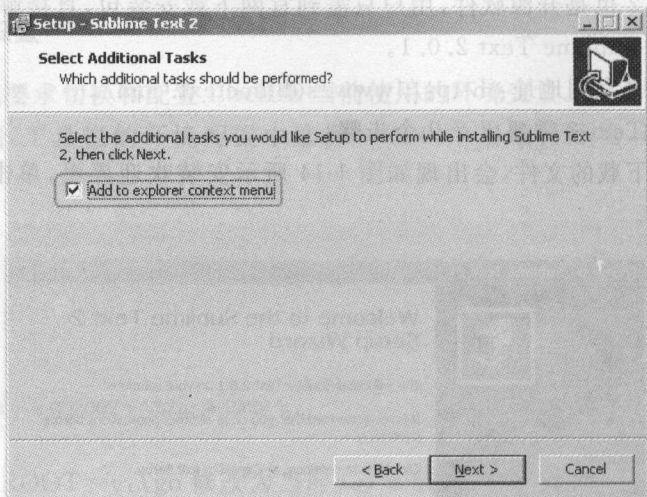


图 1-16 接受 Sublime 许可协议

(4) 进入准备安装界面后,直接单击 Install 按钮(图 1-17),进入安装过程界面。

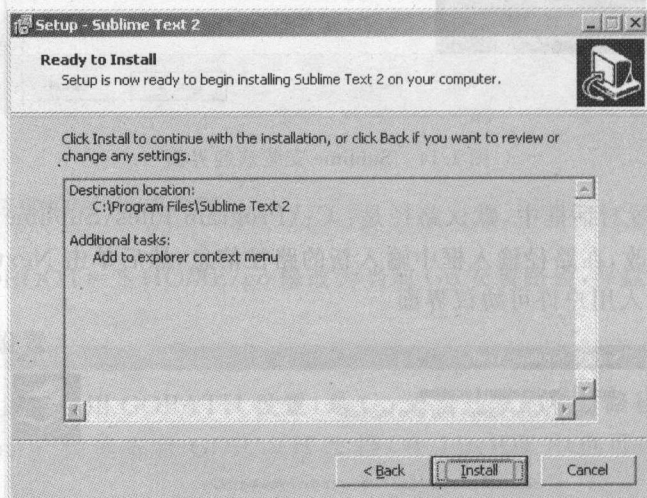


图 1-17 准备安装界面

(5) 等待安装结束,进入安装结束界面,单击 Finish 按钮(图 1-18),完成安装。

2. 安装包控制插件 Package Control

Sublime Text 2 拥有良好的扩展功能,这就是安装包(Package),Sublime Text 2 通过安装包控制插件 Sublime Package Control 就能很轻松地实现对 Package 的安装、升级和卸载。Sublime Text 2 刚安装好时在 Preferences 菜单栏中并没有 Sublime Package Control 插件,如图 1-19 所示。

Sublime Package Control 插件需用户手工安装,步骤如下。

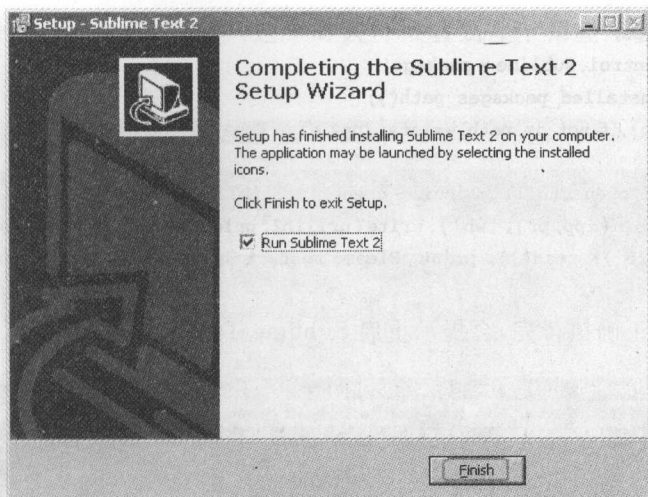


图 1-18 Sublime 安装结束

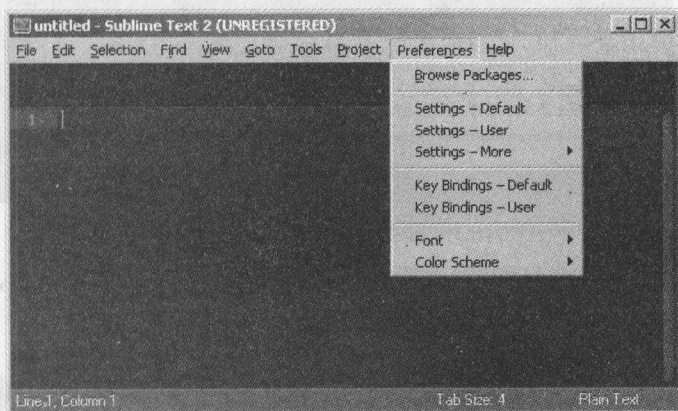


图 1-19 Sublime 刚安装好时没有包控制插件

(1) 运行 Sublime Text 2,按 Ctrl+'键打开命令行,将以下脚本代码粘贴进命令行并回车,如图 1-20 所示。

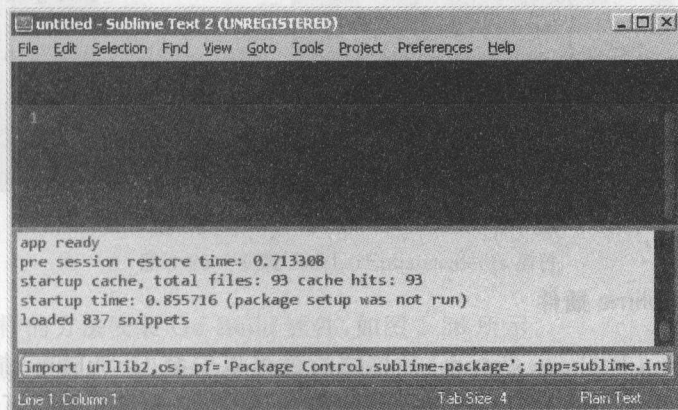


图 1-20 在命令行中执行脚本

```
import urllib2,os;
pf = 'Package Control.sublime-package';
ipp = sublime.installed_packages_path();
os.makedirs(ipp) if not os.path.exists(ipp) else None;

urllib2.install_opener(urllib2.build_opener(urllib2.ProxyHandler()));
open(os.path.join(ipp,pf), 'wb').write(urllib2.urlopen('http://sublime.wbond.net/' + pf.
replace(' ','%20')).read()); print 'Please restart Sublime Text to finish installation'
```

(2) 脚本代码正确执行完,会提示重启 Sublime Text 2,如图 1-21 所示。

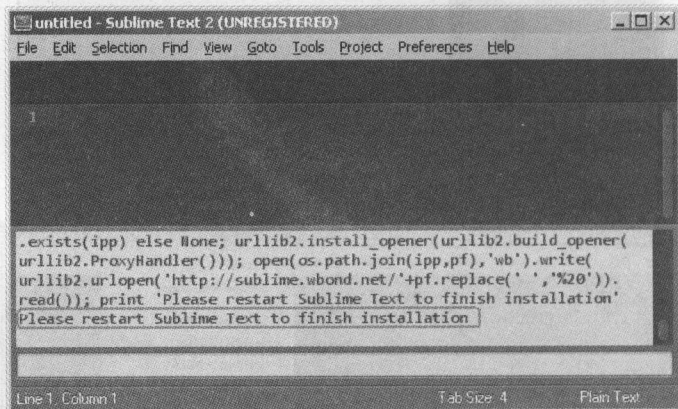


图 1-21 脚本正确执行重启 Sublime

(3) 重启 Sublime Text 2 后,如果在 Preferences 菜单栏出现 Package Settings、Package Control 菜单项,就说明安装成功了,如图 1-22 所示。

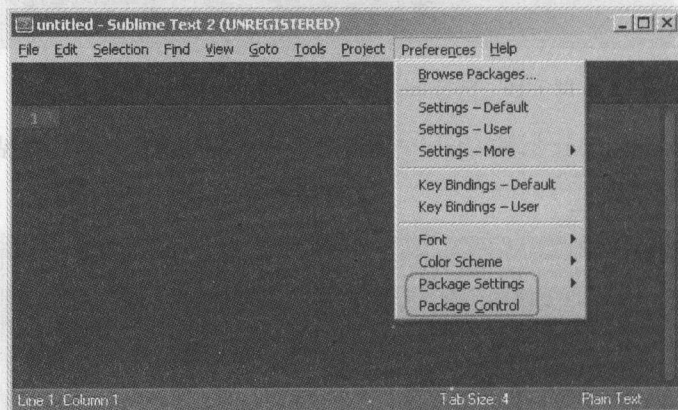


图 1-22 Sublime Package Control 安装成功

3. 安装 GoSublime 插件

安装完 Sublime Package Control 插件之后就可以安装 GoSublime 插件了,共需安装三个插件: GoSublime、SidebarEnhancements 和 Go Build。安装步骤如下。

(1) 按 Shift+Ctrl+P 键打开 Package Control 命令界面,输入“pcip”(即“Package Control:

Install Package”的缩写)命令并回车,这个时候可以看到左下角显示正在读取包数据,如图 1-23 所示。

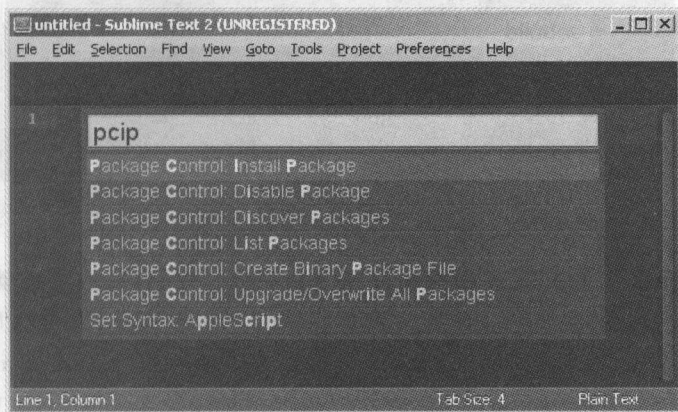


图 1-23 执行 pcip 命令

(2) pcip 命令执行完成后,出现如图 1-24 所示界面,这个时候输入“gosublime”,回车就开始安装 GoSublime 插件了。

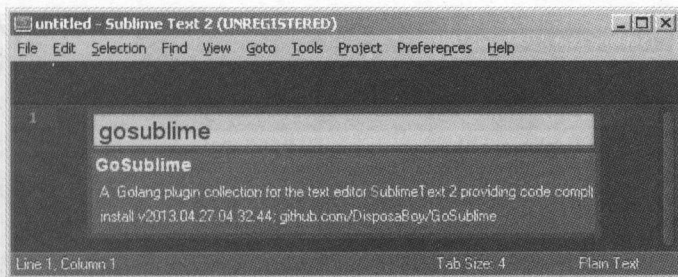


图 1-24 安装 GoSublime 插件

(3) 使用同样的方法安装 SidebarEnhancements 插件,如图 1-25 所示。

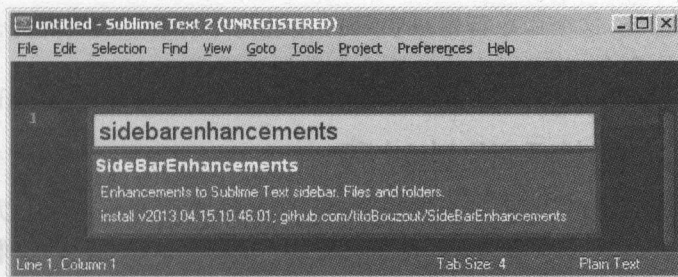


图 1-25 安装 SidebarEnhancements 插件

(4) 使用同样的方法安装 Go Build 插件,如图 1-26 所示。

安装插件之后要重启 Sublime 生效,这时就可以正式使用 Sublime Text 2 设计 Go 程序了。为了验证是否安装成功,可以打开 Sublime,打开 main.go,看看语法是不是高亮了,输入“import”后是不是自动化提示了,import “fmt”之后,输入“fmt.”是不是自动化提示有

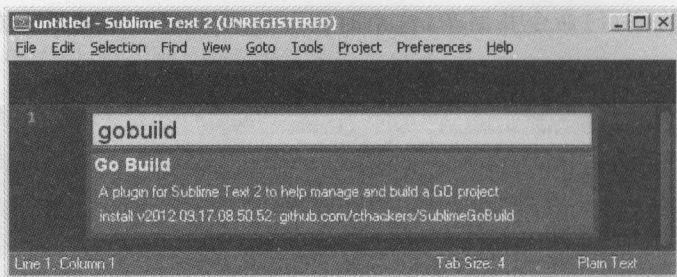


图 1-26 安装 Go Build 插件

函数了。如果已经出现这个提示,那说明已经安装完成了,并且完成了自动提示。如果没有出现这样的提示,一般就是 \$PATH 没有配置正确。可以打开终端,输入“gocode”,查看是否能够正确运行,如果不行就说明 \$PATH 没有配置正确。

4. 安装 Sublime REPL

有时用户在调试运行程序时需要输入参数,而 Sublime Text 2 的内置命令行却不支持在调试运行程序时输入参数,解决的办法是安装 Sublime REPL 插件。安装完成后会在 Tools 菜单栏的最下面生成 SublimeREPL 选项,如图 1-27 所示。

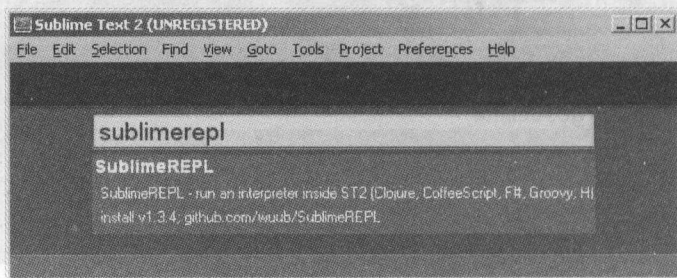


图 1-27 安装 SublimeREPL 插件

单击 Tools 菜单→选择 SublimeREPL 选项→运行 shell 命令,就可以打开命令行窗口。如果是 Windows 系统,这个命令行窗口和 cmd 系统命令行窗口的功能完全一样,可以执行所有 Windows 系统命令,并可以方便地输入参数调试程序。

1.5 Go 程序结构和设计过程

安装好 Go,并配置了环境变量,就可以使用 Sublime Text 2 编写 Go 程序了,和其他编程语言一样,本书也将通过 Hello World 的 Go 语言版,来介绍 Go 程序的结构、源代码的编辑和编译运行过程。

1.5.1 Go 程序结构

Go 程序是以包(package)的形式来组织的,这和 Python 语言类似。Go 程序一般由三部分组成:包声明部分、第三方包导入部分和函数声明部分。Go 语言使用关键字“package”声

明要创建的包；使用关键字“import”导入第三方包；使用关键词“func”声明要创建的函数。关于 Go 语言内置关键字的说明，可以参见附录 A 中的内容。

1. 本地包声明

包是组成 Go 程序的基本单位，所以每个 Go 程序源代码的开始都是一个包声明 (Package Declaration)，格式如下：

```
package <pkgName>
```

包声明的关键字是“package”，pkgName 告诉编译器当前文件属于哪个包。当要生成一个可执行的 Go 程序时，必须建立一个名为 main 的包，并且在该包中应包含一个名为 main 的函数，main 函数是 Go 程序运行时的入口。

除了 main 包之外，其他名称的包里不能包含 main 函数，且编译后都会生成 *.a 文件（也就是包文件），并被自动存放在 pkg 文件夹中。需要注意的是，一个可执行程序，有且仅有一个 main 包，一个 main 包有且仅有一个 main 函数。

2. 第三方包导入

在包声明之后，如果需要调用 Go 标准库函数（这些标准库函数通常封装在第三方包中），还要使用关键字“import”导入第三方包，这就是包的导入 (Package Import)，但必须是非 main 包。在 Go 语言中，包的导入有三种模式：正常模式、别名模式和简便模式。关于 Go 标准包的内容，可以参见附录 C 中的说明。

(1) 正常模式。在正常模式中，第三方包导入格式如下：

```
import <pkgName>
```

import 是包声明关键字，pkgName 告诉编译器要导入哪个包。包导入之后，就可以使用下面的语句格式，对包中的函数或类型进行调用：

```
<pkgName>.<funcName>
```

本例要调用 fmt 包的 Println 方法输出信息，所以调用语句为：

```
fmt.Println(...)
```

(2) 别名模式。在别名模式中，当要使用第三方包时，包名可能会非常接近或者相同，此时就可以使用别名来进行区别和调用，导入格式如下：

```
import 别名 <pkgName>
```

(3) 简便模式。在简便模式中，可以直接使用 funcName 进行第三方包的调用，不需要 pkgName，导入格式如下：

```
import . <pkgName>
```

简便模式和别名模式同时使用时容易混淆,不建议使用。import 语句在其他编程语言中都有类似的概念,比如在 C 中,“包”的概念被称为“库”,调用时使用 include 语句包含到 C 程序中。

需要注意的是,如果导入一个包之后,源程序并未调用包里的函数或类型,则编译时会报错。这其实是 Go 语言实现快速编译的一种机制,Go 在编译源文件之前,编译器会对所有导入的包进行检查,看它是否和源文件之间有关联,如果没有关联就是没用的,则不允许导入。这样可以保障源代码体积是最小的,也可以加快编译速度。

3. 函数声明

和其他高级程序设计语言一样(比如 C 语言),Go 程序的真正执行体也是函数,所以每个 Go 程序至少应声明一个 main 函数。函数声明以关键字 func 开头,格式如下:

```
func funcName(参数列表)(返回值列表){  
    //函数体  
}
```

与其他函数不同,Go 程序中的 main 函数既没有参数,也没有返回值。如果在命令行中要向 main 传入参数,可在 os.Args 变量中保存。另外,Go 函数还支持多返回值,这将在后面的章节中讲到。

1.5.2 Go 程序设计过程

在了解 Go 程序结构的基础上,就可以着手编写 Hello World 的 Go 源程序文件了,初学者可以按照以下步骤来进行编写:

(1) 在工作目录(本教程测试用工作目录是 D:\go\development)的 src 文件夹中创建源文件文件 helloworld.go,注意扩展名是“.go”。

(2) 直接将 helloworld.go 拖入 Sublime Text 2,就可以进行编辑了,输入如图 1-28 所示的代码,按 Ctrl+S 键进行存盘。

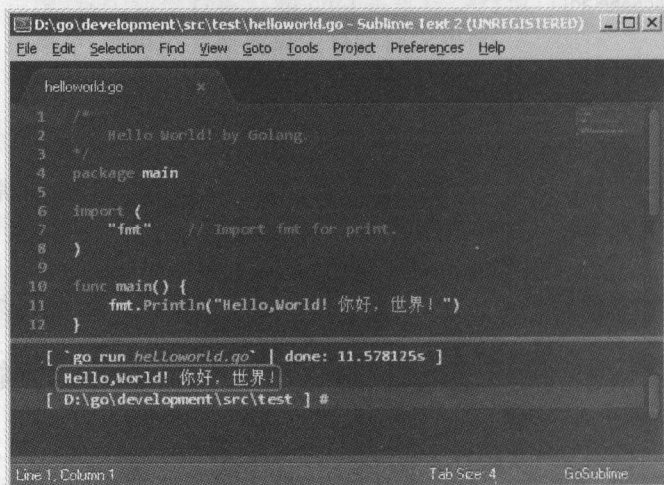


图 1-28 Hello World 程序 Go 语言版

(3) 按 Shift+Ctrl+B 组合键,打开 Sublime 的内置命令行界面,文件访问路径会自动跳转到当前目录下,然后执行如下命令:

```
go run helloworld.go
```

如果没有语法错误,会出现如图 1-28 所示的结果,可以看出中文显示正常,说明 Go 是完全支持 UTF-8 的。

1.5.3 Go 源程序语法要点

通过对 helloworld.go 源代码进行分析,发现 Go 语言程序有一些语法要点,值得初学者注意。主要有 Go 语句中的分号、注释方式、左大括号、对 UTF-8 的支持等。

1. Go 语句中的分号

通过 helloworld.go 还可以看到 Go 程序语句里没有分号。其实,和 C 语言一样,Go 语言的正式语法也使用分号来终止语句。和 C 语言不同的是,这些分号由词法分析器在扫描源代码过程中使用简单的规则自动插入,因此输入源代码多数时候就不需要分号了。Go 程序中,出现分号的典型位置是在 for 循环的分隔语句或类似之处。

2. 左大括号约定

Go 语言规定,函数、控制结构(if、for、switch 或 select)等的左大括号“{”,“必须和函数声明或控制结构放在同一行”。如果将左大括号“放在函数声明,控制语句的下一行”,编译器会在左大括号的前方自动插入一个分号,这可能导致异常的结果。

3. Go 语言对 UTF-8 的支持

最后可以看到输出的内容里面包含中文“你好,世界!”,实际上,Go 是天生支持 UTF-8 的,任何字符都可以直接输出,甚至可以用 UTF-8 中的任何字符作为标识符,这是因为 UTF-8 的发明者也就是 Go 的发明者,所以它天生就具有多语言的支持。

4. 可见性规则

Go 语言中,使用大小写来决定常量、变量、类型、接口、结构或函数是否可以被外部包所调用,根据约定:

(1) 函数名首字母小写,即为 private。

(2) 函数名首字母大写,即为 public。

1.5.4 Go 的注释方式

在 Go 程序代码中添加注释,是为了让代码阅读起来更容易理解,良好的编码习惯其中就包括为代码加上清晰明确的注释说明。Go 程序的注释方式有两种:单行注释和块注释。

1. 单行注释

单行注释以“//”开始,一般对单行语句的含义进行说明,可以单独放在一行,也可以跟

在代码后,和代码放在同一行。下面是单行注释的例子。

```
//每个 main 包有且仅有一个 main 函数
func main() {
    fmt.Println("Hello World! 你好,世界!") //函数 Println()用于行输出
}
```

2. 块注释

块注释以“/*”开始,以“*/”结束,经常用于对一个函数或语句块进行说明。下面是块注释的例子。

```
/*
    描述: 函数 main()是一个可执行包的入口函数
    参数: 无
    返回值: 无
*/
func main() {
    fmt.Println("Hello World! 你好,世界!")
}
```

3. 使用注释限制代码执行

注释的一般作用是代码添加阅读说明,但有时也可以使用注释限制某些语句的执行,使编译器不去处理它。尤其在调试程序的时候非常有用。例如:

```
func main() {
    t := time.Now()           //时间戳
    //fmt.Println(t)
/*
    fmt.Println(t.String())    //年月日时分秒
    fmt.Println(t.Format("2006year 01month 02day")) //年月日
*/
    fmt.Println(t.Weekday().String()) //星期几
}
```

上例中第3条语句前使用单行注释,表示该句被注释了,编译时编译器忽略它。第5、6条语句使用了块注释,表示这两条语句都被注释了,编译时编译器会将这两条语句都忽略掉。

小结

本章简要介绍了 Go 语言的发展与背景,Go 有哪些显著特性值得关注;同时介绍了 Go 的主要发行版本,如何获取,不同版本适合什么样的操作系统平台和 CPU 架构;重点介绍了 Go 的安装和环境变量配置;另外,还介绍了一个 Go 程序的集成开发工具 Sublime Text

2;最后通过 Hello World 了解了 Go 程序的一般结构。

通过这一章的学习,首先要学会选择一个合适的 Go 版本,下载并安装在自己的机器上,能正确配置 Go 运行的环境变量,掌握常用的 Go 命令,会查看 Go 文档。最后,要学会使用 Sublime Text 2 编写 helloworld.go 程序,调试并运行。

习题

- 1.1 浏览 Golang 官方网站,写出 Go 语言的主要特点。
- 1.2 根据自己的系统平台下载并安装 Go,配置环境变量。
- 1.3 下载并安装 Sublime Text 2,安装 GoSublime、SidebarEnhancements 和 Go Build 插件。
- 1.4 Go 程序是以_____的形式来组织的。
 - A. 函数
 - B. 类
 - C. 包
 - D. 组件
- 1.5 编写第一个 Go 程序 HelloWorld.go,熟悉 Go 编程环境及过程。

在程序运行过程中,其值可以改变的量为变量(Variable)。变量代表内存中具有特定属性的一个存储单元,它用来存放数据,也就是变量值(Variable-value)。在程序运行期间,这些值是可以改变的。

每个变量都应该有一个名字,称为变量名,即标识符。标识符是变量名,常用来标识变量。所有标识符的标识符(Identifier),它实际上代表变量在内存中的地址,可以使用取地址符“&”获取。

在内存中的地址,可以使用取地址符“&”获取。标识符是变量名,常用来标识变量。所有标识符的标识符(Identifier),它实际上代表变量在内存中的地址,可以使用取地址符“&”获取。

标识符是变量名,常用来标识变量。所有标识符的标识符(Identifier),它实际上代表变量在内存中的地址,可以使用取地址符“&”获取。

标识符是变量名,常用来标识变量。所有标识符的标识符(Identifier),它实际上代表变量在内存中的地址,可以使用取地址符“&”获取。

标识符是变量名,常用来标识变量。所有标识符的标识符(Identifier),它实际上代表变量在内存中的地址,可以使用取地址符“&”获取。

标识符是变量名,常用来标识变量。所有标识符的标识符(Identifier),它实际上代表变量在内存中的地址,可以使用取地址符“&”获取。

第2章

Go 数据类型、运算符与表达式

2. 块注释

计算机程序处理的是数据对象,不同的数据对象有着不同的存储、处理和运算方式。Go 语言提供了非常丰富的数据类型,包括基本数据类型、值类型、引用类型、接口类型、错误类型和函数类型。这些数据类型中有些也是其他程序设计语言中常见的,比如:整型、浮点型、字符型等;有些是 Go 语言特有的,比如:切片、通道、接口等;还有些其他语言中也有,但使用方式完全不同,比如:布尔型、函数型等。对于这些 Go 语言特有的和使用方式不同的数据类型,学习时要特别注意。

2.1 常量、变量与命名规则

在应用程序执行期间,常量用来表示固定不变的数据,而变量用来存储可能变化的数据。通常,在使用常量和变量之前要进行声明。

2.1.1 常量

在程序运行过程中,其值不能改变的量称为常量(Constant)。在 Go 语言中,通常用标识符代表一个常量的符号,称为符号常量(Symbolic Constant)。常量在形式上和变量有些相似,但不能像变量那样在代码中被修改,或对其重新赋值。

1. 常量声明

Go 语言使用关键字“const”声明常量,需要指定常量名称和常量数据类型,常量一般声明格式如下:

```
const <constName> [constType] = <赋值表达式>
```

在上述语句中,常量名称使用标识符(Identifier)命名,常量类型可指定基本的数据类型,常量值可以直接指定,也可以由赋值表达式生成。但应注意的是,常量值必须是能够在编译时就确定的值。另外,常量的赋值表达式可以涉及计算过程,但是所有用于计算的值必须在编译期间就能获得。

例如,声明常量 PI:

```
const PI float32 = 3.1415926
```

在进行常量声明时,也可以隐式类型定义,即可以省略常量类型。在这种方式下,编译器会根据赋值表达式产生的值,自动推断常量的类型,比如上例还可以声明为:

```
const PI = 3.1415926
```

2. 常量声明过程中的注意事项

在 Go 语言中声明和使用常量时,应注意以下事项:常量的值在编译时就已经确定,在运行时不能改变常量的值;常量的定义格式与变量基本相同,可以单个定义,也可以多个定义,比如常量组,这在后面的内容中会讲到;等号右侧必须是常量或者常量表达式,因为只有常量表达式的值在编译时才是确定的,如果是运行时才给常量赋值,编译时就会报错;常量表达式中的函数必须是 Go 语言内置函数,不能是用户自定义函数,或是从其他包中导入的函数,因为只有内置函数在编译之前才是确定的。

2.1.2 变量

在程序运行过程中,其值可以改变的量称为变量(Variable)。变量代表内存中具有特定属性的一个存储单元,它用来存放数据,也就是变量值(Variable-value),在程序运行期间,这些值是可以改变的。

1. 变量的内存模型

每一个变量都应该有一个名字,称为变量名(Variable-name)。变量名是用来引用变量所存储数据的标识符(Identifier),它实际上代表变量在内存中的地址,可以使用取地址符“&”获取。

使用变量时要注意区分变量值与变量名这两个不同概念,如图 2-1 所示。当程序编译时,编译器会自动给每一个变量名分配对应数据类型的内存地址。从变量中取值,就是通过变量名找到相应的内存地址,再从该存储单元中读取数据。

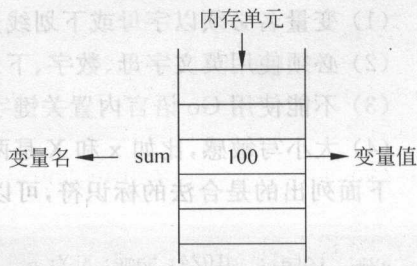


图 2-1 变量内存模型

2. 变量声明

Go 语言使用关键字“var”来声明变量,需要指定变量名称和变量数据类型,变量一般声明格式如下。

```
var < variableName > [ variableType ]
```

和常量名称一样,变量名称也使用标识符(identifier)指定,变量类型可以是 Go 语言的任何数据类型,例如,声明变量 count 为整型:

```
var count int
```

Go 语言中还可以在声明变量的同时使用“=”给变量赋初值,比如上例在声明整型变量

count 的同时给它赋初值 10, 语句如下:

```
var count int = 10
```

上例中的变量类型 type 也可以省略, 由编译器根据表达式产生的值, 自动推断变量的类型。上例还可以为:

```
var count = 10
```

Go 语言在声明变量时还允许省略掉关键字“var”, 而用“:”取代, 比如上例可以进一步简写为:

```
count: = 10
```

2.1.3 标识符与命名规则

和其他程序设计语言一样, Go 语言中用来对符号常量、变量、函数、数组、切片、通道等数据对象命名的有效字符序列统称为标识符(Identifier)。简单来说, 标识符就是数据对象的名字。

1. Go 语言命名规则

在 Go 语言中, 不管是常量的命名还是变量的命名, 都要遵循标识符命名规则:

- (1) 变量名必须以字母或下划线开始。
- (2) 必须使用英文字母、数字、下划线组成, 不能出现空格或制表符。
- (3) 不能使用 Go 语言内置关键字与保留字, 如 go、goto、break 等。
- (4) 大小写敏感, 比如 x 和 X 是两个不一样的标识符。

下面列出的是合法的标识符, 可以作为常量或变量的名称:

```
sum; _total; id1024; Name; KEY; α
```

下面是不合法的标识符, 不能作为常量或变量的名称:

```
M.D; #123; 3D64; a<b
```

2. Go 语言内置关键字

关键字对于 Go 语言有着特别的含义, 它可标识程序的结构和功能。所以, 在编写代码时, 不能用它们作为变量名或常量名, 也不能作为函数名来使用。Go 语言共有 25 个内置关键字(Built-in Keyword), 全部使用小写: break、default、func、interface、select、case、defer、go、map、struct、chan、else、goto、package、switch、const、fallthrough、if、range、type、continue、for、import、return、var。

2.2 基本数据类型

Go 语言基本数据类型(Data-types)包括布尔型、整型、浮点型、复数型、字符型、指针类型,其中复数类型是 Go 语言新增的数据类型。

2.2.1 布尔型数据

Go 语言的布尔型数据(Boolean)和其他编程语言一样,使用关键字“bool”定义。布尔型数据在内存中的字节长度为 1,取值只能是“true”和“false”,这和其他编程语言有所不同。比如有些编程语言可以用“零”来代替“false”,用“非零”代替“true”,Go 语言则不允许这么做。

例 2-1 bool 型变量的定义与使用。

```
1 //bool 型变量的定义与使用
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var b1,b2 bool
10    b1 = true
11    b2 = false
12    fmt.Println(b1)
13    fmt.Println(b2)
14 }
```

编译并运行该程序,输出结果为:

```
true
false
```

在使用布尔型数据时,需要注意以下两点。

(1) 布尔类型不接受其他数据类型赋值,如上例中让 `b1=1`; `b2=0`,编译时会出现错误提示:

```
cannot use 1 (type int) as type bool in assignment
```

(2) 布尔类型不能进行强制类型转换,比如 `b1=bool(1)`,编译时会出现错误提示:

```
cannot convert 1 (type int) to type bool
```

2.2.2 整型数据

整型数据(Integer)是计算机中的基本数据类型,每一种编程语言都会定义整型数据类型,Go语言也定义了多种整数类型。

1. 整型数据的表示方法

在Go语言中,整型数据可以用十进制、八进制、十六进制、指数形式来表示。

- (1) 十进制,如-123、0、256。
- (2) 八进制,以“0”开头的数是八进制数,如0123表示八进制数123,即十进制的83。
- (3) 十六进制,以“0x”开头的数是十六进制数,如0x123表示十六进制数123,即十进制的291。
- (4) 指数形式,由数字和字母e组成,如1e3或10e2都代表十进制的1000。

2. 整型数据的类型

Go语言共定义了10种整数类型,如表2-1所示。这些整数类型按照数据位数不同,被分为基本型和精确控制型;按照是否带符号,被分为有符号整型和无符号整型。

在Go语言中int型是有符号基本型整数,uint型是无符号基本型整数,Go语言会根据操作系统平台决定基本型整数是多少位的。如果是32位操作系统,基本型整数就是32位的;如果是64位操作系统,基本型整数则是64位的。

Go语言的精确控制型整数数共有8种:int8/uint8、int16/uint16、int32/uint32和int64/uint64,分别表示8位整数、16位整数、32位整数和64位整数。带“u”的表示无符号整数,不带“u”的表示有符号整数。

表 2-1 整数类型

类 型		字节长度	取值范围
有符号整型	int	4/8	如果是32位系统即int32,如果是64位系统即int64
	int8	1	-128~127 即 $-2^7 \sim (2^7 - 1)$
	int16	2	-32 768~32 767 即 $-2^{15} \sim (2^{15} - 1)$
	int32	4	-2 147 483 648~2 147 483 647 即 $-2^{31} \sim (2^{31} - 1)$
	int64	8	$-2^{63} \sim (2^{63} - 1)$
无符号整型	uint	4/8	如果是32位系统即uint32,如果是64位系统即uint64
	uint8	1	0~255 即 $0 \sim (2^8 - 1)$
	uint16	2	0~65 535 即 $0 \sim (2^{16} - 1)$
	uint32	4	0~4 294 967 295 即 $0 \sim (2^{32} - 1)$
	uint64	8	$0 \sim (2^{64} - 1)$

例 2-2 整型变量的定义与使用。

```
1 //整型变量的定义与使用
2 package main
3
4 import(
```

```
5     "fmt"
6 )
7
8 func main() {
9     var sum int
10    var a int = 100
11    b := 28
12    sum = a + b
13    fmt.Println(sum)
14 }
```

编译并运行该程序,输出结果为:

```
128
```

3. 整型数据的溢出

由表 2-1 可以看出,一个 int8 型整数的最大值是 127,如果再加 1,会出现什么情况? 通过下面的程序例子来观察。

例 2-3 整型数据的溢出。

```
1 //整型数据的溢出
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var a int8 = 127
10    fmt.Println(a, a + 1)
11 }
```

编译并运行该程序,输出结果为:

```
127 -128
```

通过程序运行结果发现,127+1 的运算结果不是 128,而是一128。产生这种错误的原因是整型变量在运算时,其结果超过了它的最大值,发生了“溢出”。而当这种“溢出”发生时,编译器往往不会报错,这就造成了程序数据安全隐患,需要靠程序员的细心和经验来保证运算结果的正确。

4. 整型数据溢出检查

在 Go 语言的 math 包中,有一些以“Min”或“Max”开头,后面跟上类型名的常量,这些常量记录了这种类型数的最小值和最大值。这些常量可以帮助程序员检查溢出,避免溢出的发生。

例 2-4 使用 math 包查看整型数据的最大值、最小值。

```
1 //使用 math 包查看整型数据的最大值、最小值
2 package main
3
4 import(
5     "fmt"
6     "math"
7 )
8
9 func main() {
10     fmt.Printf("Min(int8) = %v, Max(int8) = %v\n", math.MinInt8, math.MaxInt8)
11     fmt.Printf("Min(int16) = %v, Max(int16) = %v\n", math.MinInt16, math.MaxInt16)
12     fmt.Printf("Min(int32) = %v, Max(int32) = %v\n", math.MinInt32, math.MaxInt32)
13 }
```

编译并运行该程序,输出结果为:

```
Min(int8) = -128, Max(int8) = 127
Min(int16) = -32768, Max(int16) = 32767
Min(int32) = -2147483648, Max(int32) = 2147483647
```

2.2.3 浮点型数据

浮点型数据(Floating-point-number),也称为实数(Real-number),可存储带有小数的数值。浮点型数据在内存中的存储形式和整数不同,是按照指数形式存储的,共由三部分组成:符号、尾数和指数。

1. 浮点型数据的表示方法

浮点型数据有以下两种表示形式。

- (1) 十进制小数形式:由数字和小数点组成,如 1.57、0.12、0.0 都是十进制小数形式。
- (2) 指数形式:由数字和字母“e”组成,如 1.23e3 或 12.3e2 都表示 1.23×10^3 。需要注意的是“e”之前必须有数字,且“e”后面的指数必须为整数,形如 e3、1.23e2.5 的都是不合法的指数形式。

2. 浮点型数据的类型

Go 语言的浮点型分为 32 位浮点型和 64 位浮点型,如表 2-2 所示。因为已经有了 64 位浮点型,所以 Go 语言就没有必要再定义 double 型了。32 位浮点型能精确到小数点后第 7 位,64 位浮点型可以精确到小数点后第 15 位。

表 2-2 浮点型数据

类 型	字 节 长 度	精 确 位 数
float32	4	精确到小数点后 7 位
float64	8	精确到小数点后 15 位

例 2-5 浮点型数据的定义和使用。

```
1 //浮点型数据的定义和使用
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var f1 float64 = 1.23
10    f2 := 12.3
11    fmt.Println(f2 - f1)
12 }
```

编译并运行该程序,输出结果为:

```
11.07
```

3. 浮点数的舍入误差

由于浮点型数据是由有限的存数单元组成的,因此,能提供的有效数字总是有限的。在有效位以外的数字将被舍去。由此可能会产生一些误差,例如,请分析下面的程序:

例 2-6 浮点数的舍入误差。

```
1 //浮点数的舍入误差
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var f1, f2 float32
10    f1 = 12356.789e5
11    f2 = f1 + 20
12    fmt.Println(f2)
13 }
```

编译并运行该程序,输出结果为:

```
1.2356788e+09
```

上例中输出结果为: 1.2356788e+09, 即 12356.788e5。按理运算结果应该是 $f2 > f1$, 但实际是 $f2 < f1$ 。原因是 a 的值比 20 大得多, $a+20$ 的理论值应该是 1 235 678 920, 而 float32 只能精确到小数点后 7 位, 后面的数都是无意义的, 因此并不能准确地表示该数。应避免将一个很大的数和一个很小的数直接相加或相减, 否则就会“丢失”小的数。

2.2.4 复数

复数(Complex-number)是指能写成 $a+bi$ 形式的数,这里 a 和 b 是实数, i 是虚数单位(即 -1 开根)。

1. 复数的构成和类型

在 Go 语言中,复数实际上由两个浮点数构成,一个表示实部(real),一个表示虚部(imag),这和数学上的复数表示形式基本一致。

Go 语言定义了两种复数类型: `complex64` 和 `complex128`,如表 2-3 所示。

表 2-3 复数类型

类 型	字 节 长 度
<code>complex64</code>	8
<code>complex128</code>	16

例 2-7 复数的定义和使用。

```
1 //复数的定义和使用
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var cp1, cp2 complex64
10    cp1 = 1.2 + 3.4i
11    cp2 = cp1
12    cp3 := complex(1.2, 3.4)
13    fmt.Println(cp1)
14    fmt.Println(cp2)
15    fmt.Println(cp3)
16 }
```

编译并运行该程序,输出结果为:

```
(1.2+3.4i)
(1.2+3.4i)
(1.2+3.4i)
```

该例中 `cp1` 是由两个 `float32` 构成的复数,`cp2` 是由两个 `float64` 构成的复数,`cp3` 和 `cp2` 一样。

2. 复数的实部与虚部

对于一个复数 $z=\text{complex}(x,y)$,可以通过 Go 语言内置函数 `real(z)` 获得该复数的实部,也就是 x ,也可以通过函数 `imag(z)` 获得该复数的虚部,也就是 y 。关于 Go 内置函数的

介绍,可参见附录 B 的内容。

例 2-8 获取复数的实部和虚部。

```
1 //获取复数的实部和虚部
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var cpl complex64
10    cpl = 1.2 + 3.4i
11    fmt.Println("real: ",real(cpl))
12    fmt.Println("imag: ",imag(cpl))
13 }
```

编译并运行该程序,输出结果为:

```
real: 1.2
imag: 3.4
```

2.2.5 字节型数据

Go 语言使用关键字 `byte` 定义字节型数据, `byte` 型数据在内存中的字节长度为 1, 其本质就是 `uint8` 类型, 即是 `uint8` 的别名。所以, 在使用 `byte` 型数据和使用 `uint8` 型数据时的效果是完全一致的。使用别名 (Alias) 是为了增强代码的可读性, 使用 `byte` 时很清楚就会知道是进行字节处理。

1. 字节型数据的存储形式

字节型数据在计算机中主要用来表示和存储 ASCII 码, 即处理字符。将一个字符常量存放到一个字符变量中, 实际上并不是将该字符本身存放内存单元中, 而是将该字符的 ASCII 编码存放内存单元中。例如, 字符 'A' 的 ASCII 码是十进制的 65, 字符 'B' 的 ASCII 码是十进制的 66。

2. 字节型数据的输出

既然在内存中, 字节型数据以 ASCII 码存储, 它的存储形式就与 `uint8` 整数的存储形式类似。这样, 字节型数据和 `uint8` 整数之间就可以通用。一个字节型数据既可以以字符形式输出, 也能以整数形式输出。字节型数据直接输出, 就是整数形式; 要以字符型输出, 需要先将存储单元中的 ASCII 码转换成相应的字符。

例 2-9 字节型数据的定义和使用。

```
1 //字节型数据的定义和使用
2 package main
```

```
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var ch1, ch2 byte
10    ch1 = 65
11    ch2 = 'A'
12    fmt.Println(string(ch1))    //要以字符型输出需先转换
13    fmt.Println(ch2)           //字节型数据直接输出为整数形式
14 }
```

编译并运行该程序,输出结果为:

```
A
65
```

2.2.6 rune 类型

Go 语言处理 Unicode 有个专用的数据类型 `rune`,它完全等价于 `int32`,`rune` 类型是 `int32` 类型的别名。使用别名可以增强代码的可读性,和 `byte` 类似,使用 `rune` 很清楚就会知道是进行 Unicode 字符处理。

1. Unicode 编码

随着计算机技术在世界范围内的广泛使用,国际化需求越发重要,以往的字符编码方案已经不适应现代软件开发。于是国际标准化组织(ISO)在参考很多现有编码方案的基础上,统一制定了一种可以容纳世界上所有文字和符号的字符编码方案,这就是 Unicode 编码方案。Unicode 使得在一个系统中可以同时处理和显示不同国家的文字,比如中文字符“我”对应的数字是 25105(\u6211)。

Unicode 只是将字符和数字建立了映射关系,但对于计算机而言,要存储和操作任何数据,都要用字节来表示,这其中还涉及不同计算机架构的大小端问题(Big Endian, Little Endian)。于是有了几种将 Unicode 字符数字转换成字节的方法:Unicode 字符集转换格式(UCS Transformation Format, UTF)。目前常用的 UTF 格式有以下几种。

UTF-8: 使用 1~4B 不等长方案,西文字符通常只用一个字节,而东亚字符(CJK)则需要三个字节。Go 编译器支持 UTF-8 代码格式,所以 Go 源代码中可以包含中文。

UTF-16: 用 2B 无符号整数存储 Unicode 字符,适合处理中文。

UTF-32: 用 4B 无符号整数存储 Unicode 字符,多数时候有点浪费内存空间。

2. Unicode 字符的定义和输出

Go 语言使用 `rune` 定义 Unicode 字符,Unicode 字符直接输出,就是整数形式;要以字符型输出,需要先将存储单元中的 Unicode 码转换成相应的字符。

例 2-10 Unicode 字符的定义和使用。

```
1 //unicode 字符的定义和使用
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var ch1 rune
10    ch1 = '中'
11    ch2 := 22269
12    fmt.Println(ch1)                //Unicode 字符直接输出为整数形式
13    fmt.Println(string(ch1) + string(ch2)) //要以 Unicode 字符输出需先转换
14 }
```

编译并运行该程序,输出结果为:

20013
中国

2.2.7 uintptr 类型

在 Go 语言中,uintptr 是可以保存指针的无符号整数类型,可以保存 32 位或 64 位的指针。和 int 型一样,uintptr 型也会根据操作系统决定指针位数,32 位操作系统中,uintptr 是 32 位的;64 位操作系统中,uintptr 是 64 位的。

1. 地址和指针的概念

每在程序中定义一个变量,编译器都会为这个变量分配内存单元,计算机内存单元是以字节(Byte)为单位对数据进行存储的。例如,为 byte 类型分配 1B,为 int32 分配 4B,为 float64 分配 8B。计算机内存的每一个字节都有一个编号,就是内存地址 (Memory Address)。可以使用“&”操作符获取变量的内存地址。

在 Go 语言中规定,可以在程序中定义这样一种特殊的数据类型,用它专门存放地址,这种数据类型就叫指针。

2. 变量的指针和指针变量

一个变量的地址称为该变量的指针(Pointer),例如,图 2-2 中地址 1000 是变量 i 的指针。如果有一个变量专门用来存放另一个变量的地址,则称它为指针变量。图 2-2 中 i_pointer 就是一个指针变量,它存放的是变量 i 的地址 1000。

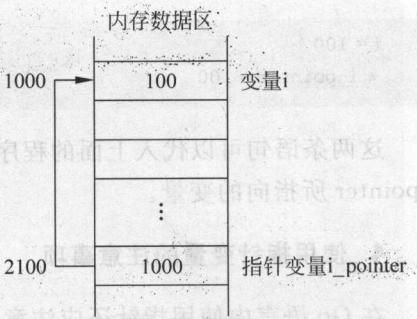


图 2-2 指针变量内存模型图

3. 指针变量的定义和引用

在 Go 语言中,为了表示指针变量和它所指向变量之间的联系,在程序中使用“*”符号表示“指向”。指针变量定义的一般形式为:

```
var <指针变量名>* <基类型>
```

运行并分析下面的程序例子。

例 2-11 指针变量的定义和引用。

```
1 //指针变量的定义和引用
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var i int           //定义整型变量 i
10    var i_pointer * int  //定义指向整型变量的指针 i_pointer
11    i = 100              //将 100 存放到 i 的内存单元中
12    i_pointer = &i       //将 i 的内存地址存放到指针变量 i_pointer 中
13    fmt.Println(*i_pointer) //通过指针变量 i_pointer 中存放的地址读取数据
14 }
```

编译并运行该程序,输出结果为:

```
100
```

通过分析上面的程序可以发现,如果已定义 `i_pointer` 为指针变量,则 `(*i_pointer)` 是 `i_pointer` 所指向的变量,如图 2-3 所示。

可以看出, `*i_pointer` 也代表一个变量,它和变量 `i` 一样指向同一个内存地址。下面两条语句作用相同:

```
i = 100
*i_pointer = 100
```

这两条语句可以代入上面的程序进行测试,第二条语句的含义是将 100 赋给指针变量 `i_pointer` 所指向的变量。

4. 使用指针变量的注意事项

在 Go 语言中使用指针还应注意以下事项:

- (1) 指针变量初始化后的默认值是 `nil`,Go 语言中没有 `NULL` 常量。
- (2) Go 语言不支持指针运算,比如 C 语言中的 `i_pointer++`、`i_pointer--` 等。

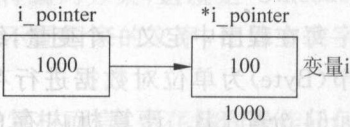


图 2-3 指针变量引用模型图

(3) Go 语言不支持“→”运算符,而是直接用“.”选择符操作指针对象成员。

2.3 运算符与表达式

一道计算机程序是由多条表达式语句按照一定的逻辑关系所构成的,表达式(Expression)是程序算法的基本单位,一条表达式又由运算符、操作数和分组符号(比如括号)共同组成。运算符(Operator),主要用于执行程序代码的运算,会针对一个或一个以上的操作数进行运算。操作数(Operand),主要定义了表达式语句中进行各种运算的量。例如,算术表达式 $2+3$,其操作数是2和3,而运算符则是“+”。

Go 语言运算符按操作数的数目可分为一元运算符、二元运算符。按照运算符的功能划分,可分为赋值运算符、算术运算符、逻辑运算符、关系运算符、位运算符等。由操作数(变量、常量、函数调用等)和运算符结合在一起构成的式子,称为表达式。对应运算符的表达式包括赋值表达式、算术表达式、逻辑表达式、关系表达式、位表达式等。

2.3.1 赋值运算符

赋值运算符由等号“=”实现,就是把等号右边表达式的值赋予等号左边的变量。等号右边的表达式可以是常量、变量、数值、函数调用等,赋值运算格式如下:

```
<变量名称> = <表达式>
```

也可以在声明变量的同时赋值,格式如下:

```
var <变量名称> [<变量类型>] = <表达式>
```

使用这种方式时,还可以省略变量类型,编译器会根据表达式产生的值,自动进行推断,从而决定变量的类型。比如表达式产生的结果是“1”,那么编译器会将变量类型设置为 int 型;又如表达式产生的结果是“1.01”,编译器则会将变量类型设置为 float 型。

例 2-12 赋值运算。

```
1 //赋值运算
2 package main
3
4 import(
5     "fmt"
6     "math"
7 )
8
9 func main() {
10     var a,b int
11     var str string = "Hello World!" //定义变量的同时赋初值
12     var pi float64
13     a = 100 //直接赋数值
```

```
14     b = len(str) //将函数 len() 的返回值赋给变量 b
15     pi = math.Pi //将 math 包中的常量 Pi 赋给变量 pi
16     fmt.Println(a)
17     fmt.Println(b)
18     fmt.Println(str)
19     fmt.Println(pi)
20 }
```

编译并运行该程序,输出结果为:

```
100
12
Hello World!
3.141592653589793
```

2.3.2 算术运算符

Go 语言的算术运算符共有 5 种,且都是二元运算符,如表 2-4 所示。由算术运算符和操作数组成的式子叫算术表达式。

表 2-4 算术运算符

运 算 符	描 述	说 明
+	加法运算符	操作数可以是整数、浮点数和字符串
-	减法运算符	操作数可以是整数和浮点数
*	乘法运算符	操作数可以是整数和浮点数
/	除法运算符	分母不能为 0,操作数为整数即是取整运算
%	取余运算符	操作数为整数,结果为整数

使用算术运算符要注意以下两点:

- (1) 乘除法优先级高于加减法,即先乘除,后加减。
- (2) 在 Go 语言中++,--是语句而非表达式。

例 2-13 算术运算。

```
1 //算术运算
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var a,b int = 7,163
10    var c,d float32 = 2.5,6.3
11    var ch1,ch2 rune = '中','国'
12    fmt.Println(string(ch1) + string(ch2))    //加法
```



```
13    fmt.Println(d-c)           //减法
14    fmt.Println(c*d)           //乘法
15    fmt.Println(b/a)           //除法
16    fmt.Println(b%a)           //取余
17 }
```

编译并运行该程序,输出结果为:

```
中国
3.8000002
15.75
23
2
```

2.3.3 关系运算符

关系运算是逻辑运算中比较简单的一种,关系运算的实质是比较运算,将两个操作数进行比较,判断其比较结果是否符合给定的条件。例如,a>10 是一个关系表达式,如果a=15,则关系表达式的值为 true; 如果 a=2,则关系表达式的值为 false。

Go 语言关系运算符共有 6 种,都是二元运算符,如表 2-5 所示。由关系运算符和操作数组成的式子叫关系表达式。

表 2-5 关系运算符

运 算 符	描 述	说 明
>	大于	二元运算符,如果 a>b,返回 true; 否则,返回 false
>=	大于等于	二元运算符,如果 a>=b,返回 true; 否则,返回 false
<	小于	二元运算符,如果 a<b,返回 true; 否则,返回 false
<=	小于等于	二元运算符,如果 a<=b,返回 true; 否则,返回 false
==	等于	二元运算符,如果 a==b,返回 true; 否则,返回 false
!=	不等于	二元运算符,如果 a!=b,返回 true; 否则,返回 false

例 2-14 关系运算。

```
1 //关系运算
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var a,b int = 5,10
10    var ch1,ch2 byte = 'a','b'
11    fmt.Println(a > b)           //大于?
12    fmt.Println(b >= 8)         //大于等于?
```

```
13     fmt.Println(ch1 < ch2)           //小于?
14     fmt.Println(ch2 <= 'B')         //小于等于?
15     fmt.Println(ch1 == 97)          //等于?
16     fmt.Println(ch2 != 98)          //不等于?
17 }
```

编译并运行该程序，输出结果为：

```
false
true
true
false
true
false
```

2.3.4 逻辑运算符

逻辑运算符用来对操作数进行基本逻辑运算，逻辑运算包括“与”、“或”、“非”，Go 语言提供了三种逻辑运算符，如表 2-6 所示。

表 2-6 逻辑运算符

运 算 符	描 述	说 明
!	逻辑非运算符	一元运算符，对操作数取反
&&	逻辑与运算符	二元运算符，对两个操作数逻辑与
	逻辑或运算符	二元运算符，对两个操作数逻辑或

逻辑运算的操作数是 bool 型，逻辑运算的结果也是 bool 型。表 2-7 是逻辑运算的“真值表”，表中 a、b 为两个操作数，其值都是 bool 型的。

表 2-7 逻辑运算真值表

a	b	! a	! b	a && b	a b
true	true	false	false	true	true
true	false	false	true	false	true
false	true	true	false	false	true
false	false	true	true	false	false

例 2-15 逻辑运算。

```
1 //逻辑运算
2 package main
3
4 import(
5     "fmt"
6 )
7
```

```
8 func main() {
9     var a,b bool = true,false
10    fmt.Println(!a)           //逻辑非
11    fmt.Println(a && b)        //逻辑与
12    fmt.Println(a || b)        //逻辑或
13 }
```

编译并运行该程序,输出结果为:

```
False
False
true
```

2.3.5 位运算符

位运算符用来对操作数的二进制位进行按位操作,Go 语言共提供了 6 种位运算符,如表 2-8 所示。位运算的操作数是整型或 byte 型,其结果也是整型或 byte 型。

表 2-8 位运算符

运 算 符	描 述	说 明
^	取反	一元运算符,对操作数按位取反
&	按位与运算符	二元运算符,对两个操作数按位与
	按位或运算符	二元运算符,对两个操作数按位或
^	按位异或运算符	二元运算符,对两个操作数按位异或
&^	标志位清除运算符	二元运算符,从 a 上清除 b 上的标志位
<<	左移运算符	二元运算符,操作数 a 向左移 b 位
>>	右移运算符	二元运算符,操作数 a 向右移 b 位

例 2-16 位运算。

```
1 //位运算
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var a,b byte = 6,11
10    fmt.Println(^a)           //取反运算
11    fmt.Println(a & b)        //按位与
12    fmt.Println(a | b)        //按位或
13    fmt.Println(a &^ b)       //标志位清除
14    fmt.Println(a << 2)       //左移 2 位
15    fmt.Println(b >> 2)       //右移 2 位
16 }
```

编译并运行该程序,输出结果为:


```
249
2
15
4
24
2
```

分析以上运算结果, $a = (6)_{10} = (00000110)_2$, $b = (11)_{10} = (00001011)_2$, 所以:

$\wedge a = (11111001)_2 = (249)_{10}$

$a \& b = (00000010)_2 = (2)_{10}$

$a | b = (00001111)_2 = (15)_{10}$

$a \& \wedge b = (00000100)_2 = (4)_{10}$

$a \ll 2 = (00011000)_2 = (24)_{10}$

$b \gg 2 = (00000010)_2 = (2)_{10}$

2.3.6 通道运算符

通道(Channel), 是 Go 语言提供的消息通信机制, 它类似于单双向数据管道(Pipe), 用户可以使用通道在两个或多个 Goroutine 之间传递消息, 通道使用运算符“<-”接收或者发送数据。

通道接收数据的格式如下:

```
ch <- value
```

通道发送数据的格式如下:

```
value = <- ch
```

在上面的操作语句中, ch 表示通道, value 表示数据, 关于通道更详尽的知识将在第 10 章讲解。

2.3.7 运算符的优先级和结合性

Go 语言规定了运算符的优先级和结合性。运算符优先级共分为 7 级, 由 7→1 优先级由高→低, 如表 2-9 所示。

表 2-9 运算符优先级

优 先 级	运 算 符	说 明
7	$\wedge !$	单目运算符
6	$* / \% \ll \gg \& \& \wedge$	双目运算符
5	$+ - ^$	双目运算符
4	$= = ! = < < = > = >$	关系运算符
3	$< -$	通道运算符
2	$\& \&$	逻辑运算符
1	$ $	逻辑运算符

Go 语言运算表达式中规定全部运算符遵循“自左向右”的结合方向,比如表达式 $a-b+c$, 先执行 $a-b$ 的运算,再执行加 c 的运算。

2.4 字符串

在 Go 语言中,字符串(String)也是一种基本数据类型,它和数组(Array)、结构体(Struct)一样都属于值类型,数组、结构体分别将在第 6 章、第 8 章讲解。

2.4.1 字符串定义

和 Go 语言相比,C/C++语言中并不存在原生的字符串类型,通常是以字符数组来表示字符串,并以字符指针来传递。Go 语言中的字符串定义和初始化非常简单,例如:

```
var str string
str = "Hello World!"
```

也可以在声明字符串变量的同时进行初始化,例如:

```
var str string = "Hello World!"
```

字符串的内容可以用类似于数组下标的方式获取,但与数组不同,字符串的内容不能在初始化后被修改(关于数组的知识可以先参考第 6 章的内容),比如对于上面的例子如果想使用语句 `str[0]='G'` 修改 `str` 的第一个字符,则编译器会报如下的错误:

```
cannot assign to str[0]
```

Go 语言还提供了一个非常有用的函数 `len()`,该函数可以用来计算字符串的长度,在实际应用程序设计过程中要经常用到 `len()` 函数处理字符串、数组、切片等。

例 2-17 字符串的定义和初始化。

```
1 //字符串的定义和初始化
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var str string
9     str = "Hello World!"
10    fmt.Println(str)
11    fmt.Println("字符串长度: ",len(str))
12 }
```

编译并运行该程序,输出结果为:

```
Hello World!  
字符串长度: 12
```

在第1章了解到,Go 编译器支持 UTF-8 的源代码文件格式。这意味着源代码中的字符串可以包含非 ANSI 的字符,比如“Hello world! 你好,世界!”可以出现在 Go 代码中。但需要注意的是,如果 Go 代码需要包含非 ANSI 字符,保存源文件时请注意编码格式必须选择 UTF-8。特别是在 Windows 下一般编辑器都默认存为本地编码,比如中国地区可能是 GBK 编码而不是 UTF-8,如果没注意这点在编译和运行时就会出现一些意料之外的情况。字符串的编码转换是处理文本文档(比如 txt、xml、html 等)非常常见的需求,不过 Go 语言仅支持 UTF-8 和 Unicode 编码。对于其他编码,Go 语言标准库并没有内置的编码转换支持。

2.4.2 字符串操作

字符串作为一种基本数据格式,也支持一些简单的运算或操作,例如可以使用“+”运算符进行字符串连接操作,可以使用 len()函数计算字符串长度等。常用的字符串操作方法如表 2-10 所示。

表 2-10 字符串操作方法

操 作	含 义	示 例	结 果
x+y	字符串连接	"Go"+"lang"	"Golang"
len(s)	计算字符串长度	len("Golang")	6
s[i]	取字符	"Golang"[1]	'o'

例 2-18 字符串的基本操作。

```
1 //字符串的基本操作  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7 func main() {  
8     var str string  
9     str = "Go" + "lang"  
10    fmt.Println(str)  
11    fmt.Println("字符串长度: ",len(str))  
12    fmt.Printf("从字符串中取字符: %c\n",str[1])  
13 }
```

编译并运行该程序,输出结果为:

```
Golang  
字符串长度: 6  
从字符串中取字符: o
```


上面仅介绍了字符串最基本的操作方法,在 Go 语言标准库的 `strings` 包中提供了更多关于字符串的操作方法,这些将在第 3 章中进行介绍。

2.4.3 字符串遍历

字符串遍历即一个一个地访问字符串中的每一个字符,Go 语言支持两种字符串遍历方式:字节数组方式遍历和 Unicode 字符方式遍历。

字节数组方式遍历,是按照下标顺序读取字符串中的每一个字符,类型为 `byte`。Unicode 字符方式遍历,也是按照下标顺序读取字符串中的每一个字符,但类型为 `rune`。

例 2-19 字符串的遍历。

```
1 //字符串的遍历
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var str string
9     str = "Hello 世界!"
10    n := len(str)
11    fmt.Println("字节数组方式遍历:")
12    for i := 0; i < n; i++{
13        ch := str[i]
14        fmt.Printf("str[%d] = %v\n", i, ch)
15    }
16    fmt.Println("Unicode 字符方式遍历:")
17    for i, ch := range str {
18        fmt.Printf("str[%d] = %v\n", i, ch)
19    }
20 }
```

编译并运行该程序,输出结果为:

字节数组方式遍历:

```
str[0] = 72
str[1] = 101
str[2] = 108
str[3] = 108
str[4] = 111
str[5] = 32
str[6] = 228
str[7] = 184
str[8] = 150
str[9] = 231
str[10] = 149
str[11] = 140
str[12] = 33
```

Unicode 字符方式遍历：

```
str[0] = 72
str[1] = 101
str[2] = 108
str[3] = 108
str[4] = 111
str[5] = 32
str[6] = 19990
str[9] = 30028
str[12] = 33
```

通过例 2-19 的运行结果可以看出，该例字符串长度为 13。尽管从直观上来说，这个字符串应该只有 9 个字符，但由于每个中文字符在 UTF-8 中占 3B，所以两个中文字符就占用了 6B。在以 Unicode 字符方式遍历时，每个字符的类型是 rune(早期的 Go 语言用 int 类型表示 Unicode 字符)，而不是 byte。

2.5 常量的初始化规则

在 2.1.1 节中介绍了常量的定义、声明和基本特点，这里将更深入地讲解常量在实际应用中的一些知识。

2.5.1 常量的类型

在 Go 语言中，分为内置常量和用户自定义常量，内置常量是由 Go 编译器定义的常量，可以直接使用。用户自定义常量由用户根据需要来建立，应先定义再使用。

1. Go 内置常量

Go 语言为了方便用户使用，提供了一些内置常量，比如：true、false、iota。true 和 false 可以直接用来初始化布尔型常量；iota 初值为 0，作为常量组中常量个数的计数器。关于常量组和 iota 的用法，将在后续内容中详细讲解。

2. 自定义常量

用户自定义常量必须是编译器能够确定的基本数值类型，比如 int 型、float 型、byte 型、complex 型、bool 型或 string 型。所以常量就分为整型常量、浮点型常量、字节型常量、复数常量、布尔型常量和字符串常量。

(1) 整型常量。整型常量可分为 int、int8、int16、int32、int64 和 uint、uint8、uint16、uint32、uint64 几种类型，可以使用八进制、十进制、十六进制三种形式来表示。整型常量默认形式是十进制，前面加“0”表示八进制形式，前面加“0x”表示十六进制形式。例如，对于 int 型常量 100，它的三种形式分别为：0144、100、0x64。

(2) 浮点型常量。浮点型常量可分为 float32 和 float64 两种类型，可以使用十进制小数形式或指数形式表示。例如，浮点型常量 123.456 的指数形式是 1.23456e2。

(3) 字节型常量。字节型常量是指 byte 类型,它实质上就是 uint8 类型,取值范围为 0~255,另外它也可以对应于 ASCII 码中的字符。例如 65 对应于字符 'A'。

(4) 复数常量。复数常量可分为 complex64 和 complex128 两种类型,它由浮点型的实部和虚部组成。对于 complex64,实部和虚部分别为 float32 型;对于 complex128,实部和虚部分别为 float64 型。例如 $1.2+3.4i$ 。

(5) 布尔型常量。布尔型常量即 bool 型,布尔型常量的值只能取 true 或 false,使用时不能误认为 0 就是 false,1 就是 true。这一点 Go 语言和其他高级程序设计语言不同。

(6) 字符串常量。字符串常量即 string 型,在初始化时字符串要使用双引号“”括起来,还可以使用内置函数 len() 获取字符串的长度,也可以按照字符串数组下标的方式取出字符串中的某个字符。例如 `len("Golang")=6`,`"Golang"[3]='a'`。

2.5.2 常量定义方法

Go 语言中的常量除了类型多样,它的定义方法也灵活多样,例如单个常量定义、多个常量一起定义以及常量组定义。

1. 单个定义

单个定义即每次只定义一个常量,例如:

```
const a int = 100
```

上述语句是单个常量定义的标准方法,“const”关键字指明要定义常量,a 是常量名,int 指明常量的类型为整型,等号“=”后面是常量的初始值,这里是 100。

另外,编译器还支持采用省略类型说明符的方式进行单个常量定义,例如:

```
const str = "Go"
```

采用这种方式时,编译器会根据常量的初始值自动判断常量的类型。比如该例中,常量 str 的类型为 string 型。

2. 多个一起定义

常量除了单独定义,还可以多个常量一起定义,这样可以使程序变得更简洁一些。例如:

```
const a, b, c = 1, "Go", 'c'
```

该例中常量 a、b、c 在一条语句中一起定义,将三条定义语句简化为一条,编译器可以自动识别出常量 a 是整型、b 是字符串型、c 是字节型。

3. 定义常量组

常量除了上述两种定义方法,还可以定义常量组,常量组有许多特殊的用法将在后面讲到。常量组的定义方法是用一对括号“()”将常量组中的常量组织在一起,例如:


```
const (  
    a = 100  
    b = 3.14  
    c = false  
)  
  
func main() {  
    fmt.Println(a)  
    fmt.Println(b)  
    fmt.Println(c)  
}
```

测试结果为：

```
100 3.14 false
```

该例的常量组中包含三个常量：a、b、c，分别为整型、浮点型和布尔型，初值为 100、3.14 和 false。

2.5.3 常量的初始化规则

在 2.1.1 节中了解到，在对常量初始化时只能使用常量或内置函数，而不能使用变量或运行时才能确定的函数对常量进行初始化。这是因为常量要在编译时就要确定其值，而变量只能在运行时才能确定其值。

1. 常量必须是确定存在的

在对常量进行初始化时，初始化值必须是确定存在的。不能使用变量对常量进行初始化，因为变量会在程序运行过程中发生改变，而常量在整个程序的运行过程中不能改变。

例如，编译下面的语句编译器会报错：

```
var i int = 100  
const f = i
```

编译器报错信息为：

```
const initializer i is not a constant
```

意思是 i 是一个未确定的数组，不能使用它对常量 f 进行初始化。

2. 使用已存在的常量对常量进行初始化

有时为了增加程序的可读性，也可以使用一个确定存在的常量对其他常量进行初始化，例如：

```
const i = 1  
const (  
    a = i
```

```
b = i + 1
c = i + 2
)
func main() {
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
}
```

测试结果为：

```
1 2 3
```

在该例中，由于常量 `i` 本身是确定存在的，所以使用它对常量 `a`、`b`、`c` 进行初始化是没有问题的。

3. 常量组初始化规则

在定义常量组时，如果不提供初始值，则表示将使用上行的表达式。例如：

```
const (
    a = 100
    b
    c
)
func main() {
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
}
```

测试结果为：

```
100 100 100
```

在该例中，在常量组中只有常量 `a` 有常量表达式，而常量 `b`、`c` 都没有常量表达式，那么 `b`、`c` 将默认使用 `a` 的常量表达式进行初始化，即 `b`、`c` 的初始化值都为 100。注意，这种用法仅在常量组中有效。

4. 多个定义常量组初始化规则

上面在使用常量组的初始化规则时，每一行只声明了一个常量，即单个定义。如果采用每行多个定义的方法，则各行定义的常量个数必须相等，然后才能使用常量组初始化规则。例如：

```
const (
    a, b = 1, 2
    c
)
```

上例中在声明常量组时,第一行定义了两个常量 `a` 和 `b`,而第二行只定义了一个常量 `c`。如果 `c` 要使用上行常量表达式对自己进行初始化,编译器就不知道到底是使用 `a`、`b` 谁的值对 `c` 进行处事化。编译时程序会报错:

```
extra expr in const decl
```

所以,要对多个定义的常量组正确使用初始化规则,则常量组中每一行常量的个数应该一致。例如:

```
const (
    a, b = 1, true
    c, d
)
func main() {
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
    fmt.Println(d)
}
```

测试结果为:

```
1
true
1
true
```

在该例中,常量 `c` 套用 `a` 的常量表达式对其初始化,初始值就为 `1`;常量 `d` 套用 `b` 的常量表达式对其初始化,初始值就为 `true`。

5. 使用内置函数初始化

Go 编译器还支持使用内置函数对常量进行初始化,例如:

```
const (
    a = 1.2 + 3.4i
    b = real(a)
    c = imag(a)
)
func main() {
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
}
```

测试结果为:

```
(1.2+3.4i)
1.2
3.4
```


在该例中,常量 `a` 初始化为复数 $(1.2+3.4i)$,常量 `b` 使用内置函数 `real()` 获取 `a` 的实部,常量 `c` 使用内置函数 `imag()` 获取 `a` 的虚部。关于 Go 语言内置函数更多的信息,可参阅第 7 章内容。

2.6 枚举

在 Go 语言中,枚举是指一系列相关的常量,如果一个常量只有几种可能的值,则可以将其定义为枚举类型(Enumeration-type)。说得更确切一些,所谓“枚举”就是指将常量的值一一列举出来,常量的值只限于列举出来的值的范围之内。最典型的枚举比如一个星期共有 7 天:星期一、星期二、星期三、星期四、星期五、星期六、星期天,对于星期这个枚举它就包含 7 个可能的值。

2.6.1 枚举类型的定义

Go 语言在定义枚举时和其他语言的方法不同,Go 语言中的枚举必须在常量组中定义。Go 语言是在整个包中取得所有枚举值,而不是像 C# 那样必须为每一个枚举定义一个名称。Go 语言在常量与枚举之间转换是非常灵活的,它在常量组中使用计数器 `iota`,`iota` 从 0 开始,常量组中每定义一个常量 `iota` 自动递增 1,通过常量组初始化规则与 `iota` 可以达到枚举的效果。

例如,对于星期这个枚举就可以定义如下:

```
const (
    Sunday      = iota //0
    Monday      = iota //1
    Tuesday     = iota //2
    Wednesday   = iota //3
    Thursday    = iota //4
    Friday      = iota //5
    Saturday    = iota //6
)
```

该例使用 `iota` 生成了从 0 开始自动增长的枚举值,即 `Sunday=0`、`Monday=1`、`Tuesday=2`、`Wednesday=3`、`Thursday=4`、`Friday=5`、`Saturday=6`。

另外,如果按行递增,可以省略后续的 `iota` 关键字。所以上例还可以定义如下:

```
const (
    Sunday      = iota //0
    Monday      //1
    Tuesday     //2
    Wednesday   //3
    Thursday    //4
    Friday      //5
    Saturday    //6
)
```

2.6.2 iota 使用规则

在使用常量组和 iota 定义枚举时,要注意 iota 的使用规则,即常量组中每定义一个常量, iota 就会自动递增 1; 而每遇到一个 const 关键字, iota 就会重置为 0。要理解好这个规则,可以通过下面的例子进行分析。

```
const (
    a = 'A'
    b
    c = iota
    d
)
const (
    e = 'E'
    f = iota
)
func main() {
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
    fmt.Println(d)
    fmt.Println(e)
    fmt.Println(f)
}
```

该例测试结果为:

```
65
65
2
3
69
1
```

该例定义了两个常量组,分析上例的运行结果:第一个常量组定义了 4 个常量,由于在常量组中每定义一个常量 iota 就会自动增 1,所以该常量组 iota 最大值为 3;第二个常量组定义了两个常量,则它的 iota 值最大为 1。所以上例的运行结果就是: c=2,因为 c 是第一个常量组的第 3 个常量; d=3,因为 d 是第一个常量组的第 4 个常量; f=1,因为 f 是第二个常量组的第 2 个常量。

2.6.3 iota 应用举例

二进制信息的基本单位是 bit(比特),而计算机信息基本存储单位是 Byte(字节),从字节开始往上按照千分位($2^{10} = 1024$)递进,分别就是 KB(KiloByte)、MB(MegaByte)、GB(GigaByte)、TB(TeraByte)、PB(PetaByte)、EB(ExaByte)、ZB(ZettaByte)和 YB

(YottaByte)。1KB是2的10次方字节,就是1024B;1YB是2的80次方,大约10的24次方个字节,完整地写下来就是1 208 925 819 614 629 174 706 176B。在例2-20中,将使用常量和iota定义计算机信息单位。

例 2-20 使用常量和iota定义计算机信息容量单位,并输出各单位的字节数。

```
1 //计算机信息容量单位
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type ByteSize float64
9
10 const (
11     _      = iota //忽略 iota = 0 的情况
12     KB ByteSize = 1 << (10 * iota)
13     MB
14     GB
15     TB
16     PB
17     EB
18     ZB
19     YB
20 )
21
22 func main() {
23     fmt.Printf("1KB = %.f Byte\n", KB)
24     fmt.Printf("1MB = %.f Byte\n", MB)
25     fmt.Printf("1GB = %.f Byte\n", GB)
26     fmt.Printf("1TB = %.f Byte\n", TB)
27     fmt.Printf("1PB = %.f Byte\n", PB)
28     fmt.Printf("1EB = %.f Byte\n", EB)
29     fmt.Printf("1ZB = %.f Byte\n", ZB)
30     fmt.Printf("1YB = %.f Byte\n", YB)
31 }
```

编译并运行该程序,输出结果为:

```
1KB = 1024 Byte
1MB = 1048576 Byte
1GB = 1073741824 Byte
1TB = 1099511627776 Byte
1PB = 1125899906842624 Byte
1EB = 1152921504606846976 Byte
1ZB = 1180591620717411303424 Byte
1YB = 1208925819614629174706176 Byte
```


在上例中,由于比较大的容量单位字节数已超出了整数的范围,所以 `ByteSize` 定义为浮点型,这样就不会造成数据溢出了。

2.7 变量的定义与声明

在 2.1.2 节介绍了变量的定义、声明和基本特点,这里将更深入地讲解变量在实际应用中的一些知识,比如变量的初始化、类型零值、类型作用域等。

2.7.1 变量的类型

变量的类型划分方式比较多,比如按照变量的数据类型进行划分,变量可分为:值类型、复合类型和引用类型。在 2.5 节了解到,在 Go 语言中只能使用常见的值类型来定义常量。而变量的类型要比常量多,除了可以使用基本的值类型定义变量,还可以使用复合类型、引用类型和一些特殊类型定义变量。

1. 值类型

值类型即基本数据类型,包括布尔型(`bool`)、整型(`int`)、浮点型(`float`)、字节型(`byte`)、复数型(`complex`)、字符串(`string`)和错误类型(`error`)。值类型的变量在传递时,编译器所做的工作是对该变量的值进行一次拷贝。

2. 复合类型

复合类型即比较复杂的值类型,包括数组(`array`)和结构体(`struct`)。数组里存放的是一组类型一致的数据,而结构体里可以存放一组不同类型的数据。

3. 引用类型

引用类型即指针类型,包括指针(`pointer`)、切片(`slice`)、字典(`map`)、通道(`channel`)、接口(`interface`)和函数(`function`)。

值类型和引用类型的区别,就在于当函数参数传递的时候:值类型是把自己的值复制一份传递给别的函数进行处理,无论复制的值怎么被改变其自身的值是不会改变的;而引用类型则是把自己的内存地址传递给别的函数进行处理,所处理的数据对象就是引用类型本身,所以随着外部的处理自己本身也被修改了,这就是传值和传址的本质区别。关于复合数据类型和引用类型,将在后续的章节中进行讲解。

2.7.2 变量的类型零值

在 Go 语言中,当一个变量被定义为某一种类型后,Go 语言会自动初始化其值为零(`Zero Value`),如表 2-11 所示。零值并不等于空值,而是当变量被声明为某种类型后的默认值。通常情况下,数值类型的默认值为“0”;布尔类型由于不能赋给数值,所以只能为“false”;字符串类型会初始化为空字符串;其他类型为“nil”,表示没有分配内存地址,使用前必须人为分配内存地址。

表 2-11 Go 语言变量初始化零值对照表

类 型	类型零值
bool	false
int,int8,int16,int32,int64	0
uint,uint8,uint16,uint32,uint64	0
float32,float64	0
complex64,complex128	0+0i
byte,rune	0
string	""
pointer,function,interface,slice,channel,map	nil

变量类型零值验证示例代码如下：

```
package main
import (
    "fmt"
)
func main() {
    var i int           //整型
    var b bool          //布尔型
    var f float32       //浮点型
    var ch byte         //字节型
    var uc rune         //Unicode
    var str string      //字符串
    var cp complex64    //复数型
    var i_pointer *int  //指针类型
    fmt.Println(i)
    fmt.Println(b)
    fmt.Println(f)
    fmt.Println(ch)
    fmt.Println(uc)
    fmt.Println(str)
    fmt.Println(cp)
    fmt.Println(i_pointer)
}
```

测试结果为：

```
0
False
0
0
0
""
(0+0i)
<nil>
```

2.7.3 变量的作用域

变量的作用范围称为变量的作用域(Scope),作用域在许多程序设计语言中非常重要。通常来说,一段程序代码中所用到的变量名并不总是有效或可用的,而限定这个名字的可用性的代码范围就是这个名字的作用域。作用域的使用提高了程序逻辑的局部性,增强了程序的可靠性,减少了名字冲突。Go语言中的变量按作用域不同可以分为两种:全局变量和局部变量。不同作用域的变量,声明位置、声明方式也不同。

1. 全局变量

全局变量(Global Variables)也称为外部变量,它是在函数外部定义的变量。它不属于哪一个函数,而是属于一个源程序文件。其作用域是整个源程序。在函数中使用全局变量,一般应作全局变量说明。只有在函数内经过说明的全局变量才能使用。但在一个函数之前定义的全局变量,在该函数内使用可不再加以说明。

在Go语言中,全局变量的作用范围涵盖整个包,从它的定义位置到包文件的结束之处。在Go语言中,全局变量使用关键字“var”进行定义,但通常放在函数之外。例如:

```
var i int
func main() {
    i = 100
    fmt.Println(i)
}
```

该例中,全局变量*i*虽然在main()函数之外定义,却能够在main()函数之内对其访问,并对其赋值。

由于全局变量从定义之处起,所有的函数都可以引用它。所以全局变量的命名最好取有特殊含义的标识符,以防止某个函数不经意地改变了它的值,使整个程序的结果出现了错误。

2. 局部变量

局部变量(Local Variables)是在函数内部定义的变量,它的作用范围仅限于函数内部,在函数内才能引用,在作用域外使用局部变量是非法的。和全局变量一样,局部变量也使用var关键字进行定义,且有时还可以省略var关键字进行最简定义(见2.6.4节)。例如:

```
func main() {
    var i int
    f()
    fmt.Println(i)
}
func f() {
    i = 200
}
```


在该例中,函数 `f()` 试图调用变量 `i` 并给它赋值,但由于变量 `i` 是由 `main()` 函数定义的局部变量,所以该例在编译时编译器提示如下错误信息:

```
undefined: i, cannot assign to i
```

意即变量 `i` 属于 `main()` 函数的局部变量,在函数 `f()` 中访问超出了变量 `i` 的作用范围。如果要在函数 `f()` 中访问 `i`,可以将 `i` 作为参数传递给函数 `f()`,这部分内容将在第 7 章进行讲解。

另外,在不同的函数中,可以定义名称相同的局部变量,它们代表不同的数据对象,互不干扰。例如:

```
func main() {  
    var a int = 100  
    fmt.Println(a)  
    f()  
}  
func f() {  
    var a float32 = 3.14  
    fmt.Println(a)  
}
```

测试结果为:

```
100  
3.14
```

该例中,在 `main()` 函数和 `f()` 中同时声明了名字为 `a` 的局部变量,且类型不同,一个是整型,另外一个为浮点型。但由于它们作用域的限制,相互之间并不会相互干扰。所以,输出结果 100 为 `main()` 函数中的变量 `a` 的值;输出结果 3.14 为函数 `f()` 中变量 `a` 的值。

2.7.4 变量的声明与赋值

在 2.1.2 节已经讲过最基本的变量的声明与赋值方法,当然在实际应用过程中,变量的声明与赋值还有许多灵活多样的方法。比如可以对单个变量进行声明与赋值,也可以同时对多个变量进行声明与赋值,本节将简要罗列一些常用变量定义方式。

1. 最标准的声明与赋值

变量最标准的声明与赋值方式由两条语句组成,这里以整型变量为例:

```
var i int  
i = 100
```

第一条语句使用 `var` 关键字定义变量 `i`,它的类型为 `int` 型;第二条语句使用等号操作符“=”为变量 `i` 赋初值 100。

这种方式主要用在,这些变量可能目前没有用到,但是会在分支结构中用到。特点是提

前声明,但赋值放在分支结构中。

2. 声明的同时赋值

变量在声明的同时还可以直接赋初值,这样在定义变量时使用一条语句就行了。所以上例还可以写成:

```
var i int = 100
```

这种方式主要用在声明全局变量当中,全局变量的赋值声明不能使用“:=”的形式,必须显式地定义类型。

3. 采用类型推断进行定义

在定义变量时,还可以省略类型说明符,这时编译器会根据变量的初值自动推断变量的类型。采用类型推断上例可以写成:

```
var i = 100
```

4. 最简方式

除了使用上面几种方式进行变量的声明与赋值,Go语言还支持一种更为简洁的方式进行变量的定义,那就是可以直接省略 var 关键字。采用最简方式上例可写成:

```
i := 100
```

观察变量声明的最简方式,发现虽然省略了 var 关键字,但等号前要加上冒号“:”,这种方式主要用在当一个函数有多个返回值时使用。

5. 多个局部变量同时声明

当需要定义多个类型一致的局部变量时,可以采用一条语句并行的方式进行声明。例如:

```
var a, b, c int = 100, 200, 300
```

该例中局部变量 a、b、c 类型相同,都是 int 型,这时就可以使用一条语句同时对它们进行声明和赋初值,这里 a=100、b=200、c=300。

6. 多个全局变量同时声明

当需要定义多个类型一致的全局变量时,首先需使用 var 关键字定义一个变量组,然后再进行单个变量的声明。例如:

```
//变量组  
var (  
    a, b, c int
```

```
)  
func main() {  
    a,b,c = 100,200,300  
    fmt.Println(a)  
    fmt.Println(b)  
    fmt.Println(c)  
}
```

测试结果为：

```
100  
200  
300
```

该例首先使用 var 关键字定义了一个全局变量组，然后在变量组中对变量 a、b、c 进行同时声明。最后在 main() 函数中对 a、b、c 进行初始化。

通过比较全局变量与局部变量的声明方式，可以发现：局部变量定义不能使用 var() 的方式进行简写，只能使用并行方式；全局变量的声明可以使用 var() 的方式进行简写，且可以使用并行方式，但不能省略 var 关键字；另外，所有变量都可以使用类型推断。

7. 赋值忽略

当进行多个变量同时定义时，还有一种特殊情况，就是需要对这些变量进行部分初始化。比如定义了 4 个整型变量 a、b、c、d，现在需要对 a、c、d 进行初始化赋值，而 b 暂时不用初始化。在这种情况下，可以使用空白符号“_”来解决这个问题。例如：

```
func main() {  
    var a,b,c,d int  
    a,_,c,d = 1,2,3,4  
    fmt.Println(a)  
    fmt.Println(b)  
    fmt.Println(c)  
    fmt.Println(d)  
}
```

测试结果为：

```
1  
0  
3  
4
```

该例共定义了 4 个整型变量 a、b、c、d，但初始化赋值语句为“a,_,c,d=1,2,3,4”，即使用了空白符号对变量 b 的赋值进行了忽略，所以变量 b 的输出结果为 0。

使用空白符号进行忽略操作，最常见的用途是选择性接收函数的返回值。Go 语言支持函数可以有多个返回值，当调用函数时如果只想接收其中一部分返回值，那么就可以使

用“_”将不需要的返回值忽略掉,关于函数的知识可以参见第7章内容。

2.8 类型别名

Go语言允许用户使用 `type` 关键字定义新的类型,这叫类型别名(Type-alias)。在2.2.5节讲到,`byte` 和 `uint8` 是等价的、可以互换,这其实是由类型别名实现的。类型别名其实什么也不做,只是给类型提供了不同的名字,让代码看起来更容易理解。比如要定义一个字“Word”,该数据是无符号的16位整型,可以使用 `uint16` 来定义。如果给 `uint16` 定义一个别名: `word`,那么 `word` 和 `uint16` 就是等价的,进而可以使用 `word` 定义用户所需要的变量,这样程序阅读起来就更加容易理解了。

2.8.1 类型别名定义方式

在Go语言中,可以使用 `type` 关键字为类型定义别名。另外,类型别名必须定义在一个类型组中,例如:

```
//类型组
type (
    word uint16           //定义 word 是 uint16 的别名
)
func main() {
    var w word = 1024
    fmt.Println(w)
}
```

测试结果为:

```
1024
```

在该例中首先使用 `type` 关键字定义了一个类型组,然后定义 `word` 为 `uint16` 的别名。在 `main()` 函数中,使用 `word` 声明了一个变量 `w` 并赋初值为 1024。

2.8.2 中文类型名

由于Go支持UTF-8编码格式,所以在定义类型别名时还可以使用非英文字符,如下例就使用了中文别名。

```
type (
    小数 float32
    文本 string
)
func main() {
    var f 小数 = 3.14
    var str 文本 = "中国"
```

```
fmt.Println(f)
fmt.Println(str)
}
```

测试结果为：

```
3.14
中国
```

在该例中，首先使用 `type` 定义一个类型组，在该类型组中定义“小数”为 `float32` 型的别名、“文本”为 `string` 型的别名。在 `main()` 函数中，使用“小数”声明变量 `f` 并赋初值为 3.14；使用“文本”声明变量 `str`，并赋初值为“中国”。

2.9 类型转换

有时由于程序功能的需要，要对变量进行类型转换 (Type Switch)。在进行类型转换时，首先要清楚 Go 语言的类型转换机制。Go 语言是类型安全的，即它不允许数据类型隐式转换，否则会造成精度丢失，所有类型转换必须显式进行。Go 的这种类型转换机制，保证了 Go 是类型安全的语言。

2.9.1 类型转换方法

变量类型转换的基本格式如下：

```
<变量 A>[:]=<变量 A 的类型>(<变量 B>)
```

上式中如果变量已经定义过了，可以省略“:”；如果定义的是一个全新的变量，则不能省略“:”。另外，Go 语言中的类型转换只能在两种相互兼容的类型之间进行。例如：

```
func main() {
    var a int
    var f float32 = 3.14
    a = int(f)           //a 已经定义过了
    b := int(f) + 1      //b 是一个全新的变量
    fmt.Println(a)
    fmt.Println(b)
}
```

测试结果为：

```
3
4
```

在该例中，浮点型变量 `f` 被显式转换成整型，并将其值赋给整型变量 `a`。在本例的转换

过程中,浮点型数据转换成整型数据只是丢失了一部分精度,但这两种数据类型本身是可以相互兼容的,所以这种转换没有问题。另外,由于 a 已经提前声明过了,所以在进行类型转换时不需要等号前面的“:”;而 b 是一个全新的变量,所以在进行类型转换时不能省略等号前面的“:”。

2.9.2 类型兼容性

所谓类型兼容性(Type-compatibility)就是指,如果两种数据类型之间可以显式转换,就说这两种类型是相互兼容的;反之,这两种类型之间是不兼容的。如果变量类型不相互兼容,在进行转换时会发生什么情况?试看下面的例子:

```
func main() {  
    var i int  
    var b bool = false  
    i = int(b)  
    fmt.Println(i)  
}
```

该例编译时编译器会报出如下错误信息:

```
cannot convert b (type bool) to type int
```

通过错误提示可以看出,整型和布尔型是相互不兼容的数据类型。这是因为在 Go 语言中规定,布尔型只有两个值: true 和 false,所以它不能够与整型数字进行相互转换。在进行类型转换时,类型间的兼容性如表 2-12 所示。

表 2-12 Go 类型兼容性

Form To \	int	float	string	[]byte	[]int
int	✗	✓	✗	✗	✗
float	✓	✗	✗	✗	✗
string	✗	✗	✗	✓	✓
[]byte	✗	✗	✓	✗	✗
[]int	✗	✗	✓	✗	✗

2.9.3 类型转换分类

在 Go 语言中,不同数据类型之间相互转换共分为三种情况:数值转换、自定义类型转换和 string 到 byte/int 的 Slice 转换。

1. 基本数据类型相互转换

Go 语言允许基本数据类型之间进行相互转换,比如整型类型之间的相互转换、浮点型之间的相互转换,整型和浮点型之间的相互转换等。

(1) 整型数据之间的相互转换,例如:

```
func main() {
    var a int = 128
    var b int8 = 127
    var c int16 = 32767
    var d int32 = 32768
    var e int64 = 32768
    fmt.Println(a, int8(a), int16(a), int32(a), int64(a))
    fmt.Println(int(b), b, int16(b), int32(b), int64(b))
    fmt.Println(int(c), int8(c), c, int32(c), int64(c))
    fmt.Println(int(d), int8(d), int16(d), d, int64(d))
    fmt.Println(int(e), int8(e), int16(e), int32(e), e)
}
```

测试结果为:

```
128 -128 128 128 128
127 127 127 127 127
32767 -1 32767 32767 32767
32768 0 -32768 32768 32768
32768 0 -32768 32768 32768
```

观察程序运行结果发现有溢出现象,这是因为如果将 int64 转换为 int8 时会发生 56 位的丢失。所以,在整型数之间转换时,最好是将数位小的数转换成数位大的数,这样就不会发生数位丢失,造成溢出现象了。

除了 5 种类型的基本整型之间可以相互转换,5 种无符号整型之间也可以相互转换。基本整型和无符号整型之间原则上也是允许相互转换的,那么转换之后会发生什么情况呢?留给读者自己试验、思考。

另外,在 2.8 节中了解到 byte 是 uint8 的别名, rune 是 int32 的别名。所以,上述整型的转换原则同样适合于这两种类型。

(2) 浮点型数据和整型数据之间的相互转换,例如:

```
func main() {
    var i, j int
    var f1 float32 = 3.14
    var f2 float64 = 123.456
    i = int(f1)
    j = int(f2)
    fmt.Println(i, j)
}
```

测试结果为:

```
3
123
```

通过观察运行结果发现, float32 和 float64 都能转换成 int 型, 只不过转换后产生了精度丢失。当然在有些应用中, 这点丢失是可以容忍的。

(3) 整型数据和浮点型数据之间的相互转换, 例如:

```
func main() {  
    var i int = 100  
    var f1 float32  
    var f2 float64  
    f1 = float32(i)  
    f2 = float64(i)  
    fmt.Printf("%f, %f\n", f1, f2)  
}
```

测试结果为:

```
100.000000,100.000000
```

2. 自定义类型相互转换

有时用户自定义类型之间的数据也可以相互转换, 例如:

```
type (  
    newint int  
    newfloat float32  
)  
func main() {  
    var i newint = 100  
    var f newfloat  
    f = newfloat(i)  
    fmt.Printf("%d, %f", i, f)  
}
```

测试结果为:

```
100,100.000000
```

通过观察运行结果可以发现, 只要用户自定义类型的底层基本类型是相互兼容的, 那么用户自定义类型之间也可以相互转换。

从严格意义上讲, 对于用户自定义类型 type newint int, 这里 newint 并不能说是 int 的别名, 而只是底层数据结构相同, 在进行类型转换时仍旧需要显式转换。但 byte、rune 确实为 int8 和 int32 的别名, 可以互换使用。

3. string 到 byte/int 的 Slice 转换

通过 2.4 节的知识了解到, 字符串也可以看作是字符数组。而在 Go 语言中, 基本字符包括 ASCII 和 Unicode。ASCII 字符通常使用 byte 类型来定义, 而 Unicode 字符通常使用

rune 类型来定义(rune 是 int 的别名)。所以 string 也可以看作是 byte/int 的切片(Slice), 即[]byte 或[]int。

既然 string 和[]byte 或[]int 是同一类事物,也就是说它们是可以相互兼容的,所以它们之间也可以相互转换。

(1) 从 string 到 byte/int Slice 的转换,例如:

```
func main() {  
    var str string = "Golang"  
    var byteSlice []byte  
    byteSlice = []byte(str)  
    for _, v := range byteSlice {  
        fmt.Printf("%c", v)  
    }  
}
```

测试结果为:

Golang

该例首先定义字符串变量 str,然后使用关键字 byte 将 str 转换成字节切片 byteSlice,最后使用 for range 语句遍历 byteSlice,并输出它的每一个元素。

(2) 从 byte/int Slice 到 string 的转换,例如:

```
func main() {  
    var str string  
    var byteSlice = []byte{'H', 'e', 'l', 'l', 'o'}  
    str = string(byteSlice)  
    fmt.Println(str)  
}
```

测试结果为:

Hello

该例首先定义并初始化字节切片 byteSlice,然后使用关键字 string 将 byteSlice 转换成字符串 str,最后输出字符串。

在 Go 语言应用程序开发中,string 和 byte/int Slice 之间的转换非常有用,关于 Slice 的知识,将在第 6 章进一步深入学习。

小结

本章主要介绍了 Go 语言的基本数据类型、运算符和表达式。Go 语言的基本数据类型包括其他高级语言里常见的整型、浮点型、布尔型、字符串等,还包括一些特有类型,比如复数型、rune 类型、uintptr 类型等。Go 语言的运算符分为赋值运算符、算术运算符、关系运算

符、逻辑运算符、位运算符、通道运算符等,其中通道运算符也是 Go 特有的。

通过这一章的学习,首先要学会使用 Go 的基本数据类型定义常量和变量,在定义常量时要灵活运用常量的初始化规则,在定义变量时要注意变量的类型零值。关于常量的另外一个应用,就是配合 iota 定义枚举类型,这和其他高级语言不同。最后就是要掌握类型别名,了解在 Go 语言中必须显式进行类型转换。

习题

2.1 指出下面不正确的变量名。_____

- A. count B. _total C. 3D64 D. id1024

2.2 写出下列整型数的八进制和十六进制形式。

- (1) 24 (2) -16 (3) 125 (4) -512

2.3 Go 语言中布尔型数据的取值只能是_____和_____。

2.4 设 $a=15, b=27$, 写出下述表达式的运算结果。

- (1) $a+=b$ (2) $b-=2$ (3) $a*=5+6$ (4) $a\%=6$

2.5 下述语句有错误,分析原因,试改正。

```
func main() {
    var a,b int = 15,6
    var f float64
    f = float64(a /b)
    fmt.Println(f)
}
```

2.6 定义虚数变量 cp,初始化其实部为 1.23、虚部为 4.56,并打印输出 cp。

2.7 在 Go 语言中,byte 类型在内存中的字节长度为_____,它是_____类型的别名; rune 类型在内存中的字节长度为_____,它是_____类型的别名。

2.8 阅读下面的程序:

```
func main() {
    var a,b int = 10,1
    var p * int
    p = &a
    a = *p + b
    fmt.Println(a)
}
```

执行该程序段后, a 的值为_____。

- A. 12 B. 11 C. 10 D. 编译出错

2.9 定义字符串变量 str1、str2,通过 scanf() 函数给这两个变量输入数据,并将它们合并之后输出。

2.10 阅读下面的程序(假设字符 'A' 的 ASCII 码值是 65):

```
const (
```

```
    a = 'A'
```

```

b = a + iota
c
d = iota
)
func main() {
    fmt.Println(a,b,c,d)
}

```

执行该程序段后, $a = \underline{\hspace{2cm}}$, $b = \underline{\hspace{2cm}}$, $c = \underline{\hspace{2cm}}$, $d = \underline{\hspace{2cm}}$ 。

(1) 赋值语句。在表述一个算法时,经常要引入变量,并赋给该变量一个值。用来表明赋给某一个变量一个具体的确定值的语句叫做赋值语句(Assignment Statement)。在算法描述中由一个赋值语句(assignment)组成的最简单位是本基的语句。Go语言下以可执行的语句为基本单位。赋值语句的格式为:变量名=表达式;。其中,变量名是合法的标识符,表达式是合法的Go表达式。赋值语句的作用是计算表达式或改变变量的值。赋值语句的左边,值应在“=”的右边。值可以是常量、变量、表达式、函数调用等。还可以用其他表达式,或是某个函数的返回值。例如:

```

//给变量赋值
name = "李明"

```

对变量赋值时,变量名写在等号左边,表达式写在等号右边。赋值语句的格式为:变量名=表达式;。其中,变量名是合法的标识符,表达式是合法的Go表达式。赋值语句的作用是计算表达式或改变变量的值。赋值语句的左边,值应在“=”的右边。值可以是常量、变量、表达式、函数调用等。还可以用其他表达式,或是某个函数的返回值。例如:

(2) 运算符表达式语句。在Go语言中,运算符表达式语句是由一个运算符表达式组成的语句。运算符表达式语句的格式为:表达式;。其中,表达式是合法的Go表达式。运算符表达式语句的作用是计算表达式或改变变量的值。运算符表达式语句的左边,值应在“=”的右边。值可以是常量、变量、表达式、函数调用等。还可以用其他表达式,或是某个函数的返回值。例如:

2. 函数调用语句

Go语言函数调用语句的格式为:函数名(参数列表);。其中,函数名是合法的标识符,参数列表是合法的Go表达式。函数调用语句的作用是调用函数。函数调用语句的左边,值应在“=”的右边。值可以是常量、变量、表达式、函数调用等。还可以用其他表达式,或是某个函数的返回值。例如:

第3章

Go 顺序结构程序设计

通过前两章的知识可以了解到,Go 程序的基本分发单位是包(Package),一个包由预处理命令(Import)、常量声明、全局变量声明和若干函数组成。一个函数又由声明部分和执行语句组成。不同类型的执行语句,可以将 Go 程序组织成:顺序结构、分支结构和循环结构。本章将介绍几种简单的 Go 语句,并使用它们设计顺序结构的 Go 程序。

3.1 顺序结构程序设计和基本语句

在进行程序设计时,一般有两部分工作要做:一部分是数据设计,另一部分是操作设计。数据设计的结果是一系列数据描述语句,主要用来定义数据类型,完成数据的初始化等(第2章主要就是这方面的知识内容)。操作设计的结果是一系列操作控制语句,其作用是向计算机系统发出操作指令,以完成对数据的加工和流程控制(本教材第3~5章主要就是介绍这方面的知识内容)。

3.1.1 顺序程序结构

计算机程序设计结构一般可分为三种类型:顺序结构、选择结构和循环结构。顺序结构(Chronological Structure)程序设计是最简单的,只要按照解决问题的顺序写出相应的语句就行,它的执行顺序是自上而下,依次执行。不过大多数情况下顺序结构都是作为程序的一部分,与其他结构一起构成一个复杂的程序,例如分支结构中的复合语句,循环结构中的循环体等。

3.1.2 简单语句

在 Go 语言中,无论是运算操作还是流程控制,都是由相应的语句完成的。组成顺序结构程序的简单语句主要包括表达式语句、输入输出语句、空语句和函数调用语句。输入输出语句将在 3.3 节专门介绍,这里仅介绍表达式语句、空语句和函数调用语句。

1. 表达式语句

由表达式组成的语句叫表达式语句。表达式语句很简单,表达式加上分号“;”就是一个表达式语句。在编辑 Go 源程序代码时,如果一行只有一条表达式语句,分号可以省略;如果一行包含多条语句,则必须使用分号将不同语句隔开。例如:


```
//一行只有一条语句分号可以省略
name := "李明"
fmt.Println("My name is ", name)
//一行有多条语句必须使用分号隔开
name1 := "李明" ; fmt.Println("My name is ", name1)
```

表达式语句可以分为赋值语句和运算符表达式语句,其作用是计算表达式或改变变量的值。

(1) 赋值语句。在表述一个算法时,经常要引入变量,并赋给该变量一个值。用来表明赋给某一个变量一个具体的确定值的语句叫做赋值语句(Assignment Statement)。在算法语句中,赋值语句是最基本的语句。

Go 语言中的赋值语句由赋值表达式后跟一个分号组成,在程序设计过程中,赋值语句应用十分广泛。在 2.3.1 节中了解到,赋值表达式由赋值运算符“=”实现,变量应在“=”的左边,值应在“=”的右边。值可以是一个常量或某种类型的数值,还可以由其他表达式产生或是某个函数的返回值。例如:

```
//将数值直接赋给变量
name := "李明"
//将一个常量赋给变量
pi := math.Pi
//将表达式结果赋给变量
sum := 100 + 200
//将函数的返回值赋给变量
strlen := len("Hello World!")
```

(2) 运算符表达式语句。在 Go 语言中,运算符表达式语句通常由运算符表达式后跟一个分号组成,例如:

```
i++;
i--
```

在该例中,“i++;”语句的功能是变量 i 的值增 1;“i--;”语句的功能是变量 i 的值减 1。这两条语句通常用于循环控制语句中,对循环控制变量进行增、减操作。

2. 函数调用语句

函数调用语句由函数调用表达式后跟一个“;”组成,其主要作用是完成特定的任务,和表达式语句一样,如果一行只有一条表达式语句,分号也可以省略。

Go 语言函数分为:内部函数、标准库函数和自定义函数,要了解函数的更多知识可先参阅第 7 章内容。内部函数是 Go 编译器自身提供的一系列函数,共有十几个(见附录 A);自定义函数是用户自己声明的函数,它只能在当前包中调用。对于内部函数和自定义函数都可以直接调用,格式如下:

```
functionName(参数列表)
```

例如：

```
//调用内置函数 len()
strlen := len("Hello World!")
//调用自定义函数 f1()
data := f1()
```

除了标准函数和自定义函数,Go 语言还提供了丰富的标准库函数,这些标准库函数都以包的形式分类组织起来(要了解 Go 语言提供的主要包和库函数可参加附录 C)。对于标准库函数,要先导入包然后才能调用,格式如下:

```
import "packageName"
packageName.functionName(参数列表)
```

例如：

```
import "fmt"
fmt.Scanf("% f",&x)
fmt.Printf("% f",x)
```

该例中标准输入、输出函数由 fmt 包提供,所以要调用这两个函数首先要导入 fmt 包。第一条语句使用 import 语句导入 fmt 包;第二条语句调用标准输入函数 Scanf(),从键盘输入 x 的值;第三条语句调用标准输出函数 Printf(),将 x 的值输出到显示器上。

3. 空语句

空语句用一个分号“;”表示,一般形式为:

```
;
```

空语句在语法上占有一条简单语句的位置,而执行该语句不调用任何操作。空语句有时用来作流程的转向点,流程从程序其他地方转到此语句处;空语句也可以用来作为循环语句中的循环体,循环体是空语句,表示循环体什么也不做,即无限循环;空语句还经常用作程序功能扩展,比如给程序预留一些功能扩展的位置,而暂时不编写程序语句代码,留作以后再扩展。所以,空语句看似简单,但也可以灵活运用。

3.1.3 复合语句

复合语句(Compound Statement)是由一对花括号“{}”将多条语句组合在一起而构成的,在语法上相当于一语句,又称为分程序。复合语句的一般形式为:

```
{
    [内部数据描述语句]
    操作语句 1
    操作语句 2
    :
}
```

操作语句 n

}

使用复合语句时要注意：

- (1) 在复合语句的“内部数据描述语句”中定义的变量是局部变量，仅在复合语句中有效。
- (2) 复合语句结束的右大括号“}”之后，不需要再加分号“；”。

例如：

```
//复合语句
package main
import (
    "fmt"
)
func main() {
    var x,y int = 1,2
    {
        var x int = 2
        {
            var x int = 3
            fmt.Println(x,y)
        }
        fmt.Println(x,y)
    }
    fmt.Println(x,y)
}
```

测试结果为：

```
3 2
2 2
1 2
```

在该例中，main 函数只有一条语句——复合语句，在这条复合语句中又嵌套了另一条复合语句。如果给它们分层，main 函数是第一层，复合语句是第二层，嵌套的复合语句是第三层。在各层中都声明了各自的局部变量 x。所以，各层引用的 x 都是各自的局部变量 x，对应不同的内存空间。因此，在各层复合语句中给 x 赋值时，不会影响其他层 x 变量的值。复合语句常用于流程控制语句中执行多条语句。

3.2 Go 程序语法注意事项

程序语句是计算机程序中一个完整的操作单位，程序主要是通过执行语句完成各种工作。程序语句会向计算机发出操作指令，来实现程序的各种功能。

要用户自己来控制。比如该例中使用 `fmt.Println("\n")` 语句，来实现一个换行输入，以

3.2.1 Go 程序语句和分号的使用

Go 语言程序语句一般分为变量声明语句、表达式语句、控制语句、函数调用语句、输入输出语句、返回语句等。

用户在编写程序时要注意,Go 程序语句末尾没有分号“;”。其实和 C 语言一样,Go 语言的正式语法也使用分号来终止语句。但不同的是,这些分号由词法分析器在扫描源代码过程中使用简单的规则自动插入,因此用户输入源代码多数时候就不需要分号了。

通常 Go 程序仅在 for 循环语句中使用分号,以此来分开计数器初始化语句、条件表达式语句和步进表达式语句。

Go 语言不允许将两条语句写到一行,如果必须要写在一行,要使用分号将这两条语句分隔开。

3.2.2 Go 程序语句块和左大括号约定

语句块就是用“{”和“}”括起来的若干条程序语句。例如,一个函数中的语句都包含在用“{”和“}”括起来的函数体中,同样还有分支结构或循环结构的语句块也被括起来。这些语句块在逻辑上都属于一个整体。

Go 语言规定,函数、控制结构(if、for、switch 或 select)的左大括号“{”,“必须和函数声明或控制结构放在同一行”。如果将左大括号“放在函数声明和控制语句的下一行”,编译器会在左大括号的前方自动插入一个分号,这可能导致异常的结果。

用“{”和“}”把相关代码括起来,有助于更清晰、更准确地定义程序逻辑边界,有助于更容易地阅读和分析程序。

3.2.3 注释语句

注释语句(Comment Statement),是在程序的开始或中间,不具有任何功能实现的作用,仅仅是对程序进行说明的语句。Go 程序的注释方式有两种:单行注释和多行(块)注释。形如:

```
//单行注释
/*
    多行(块)注释
*/
```

3.3 数据输入输出

一个计算机程序任务通常由数据输入、数据处理、数据输出三部分组成,所以数据输入、输出语句是程序中最普通的语句,几乎每一个程序都要使用它。Go 语言提供了多种用于实现数据输入、输出的函数,这些函数在 fmt 包中实现。在输入、输出数据时,首先要导入 fmt 包,导入语句为:

```
import "fmt"
```

3.3.1 标准输出函数

数据输出(Data Output),是计算机对各类输入数据进行加工处理后,将结果以用户所要求的形式输出到标准输出设备上(比如显示器)。在 Go 标准库的 `fmt` 包中,有三种标准输出函数: `Print()`、`Printf()`和 `Println()`。

1. `Print()` 函数

`Print()`函数的功能是按照系统默认格式字符串和参数表,生成一个打印字符串,然后再将其输出到标准输出设备上,输出时会在操作数之间自动加上空格。`Print()`函数执行后会返回输出的字节数或错误类型,`Print()`函数原型定义如下:

```
func Print(a ... interface{}) (n int, err error)
```

例 3-1 使用 `Print()`函数默认格式打印输出数据。

```
1 //Print()函数默认格式打印
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var id int = 100
9     var name string = "李明"
10    var grade float32 = 91.5
11    fmt.Print("默认格式打印: \n")
12    fmt.Print(" 学号: ", id)
13    fmt.Print("\n")
14    fmt.Print(" 姓名: ", name)
15    fmt.Print("\n")
16    fmt.Print(" 成绩: ", grade)
17 }
```

编译并运行该程序,输出结果为:

```
默认格式打印:
学号: 100
姓名: 李明
成绩: 91.5
```

在例 3-1 中可以看到,由于 `Print()`函数不支持格式化输出,所以像回车、换行这些操作要用用户自己来控制。比如该例中使用 `fmt.Print("\n")`语句,来实现一个换行操作。

2. Println() 函数

Println() 函数和 Print() 函数的功能基本一致,不同的是 Println() 函数会在输出结束后再自动输出一个换行,Println() 函数的原型定义如下:

```
func Println(a ...interface{}) (n int, err error)
```

例 3-1 的输出效果如果使用 Println() 函数实现,则代码如例 3-2 所示。

例 3-2 使用 Println() 函数默认格式打印输出数据。

```
1 //Println()函数默认格式打印
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var id int = 100
9     var name string = "李明"
10    var grade float32 = 91.5
11    fmt.Println("默认格式打印:")
12    fmt.Println("学号:", id)
13    fmt.Println("姓名:", name)
14    fmt.Println("成绩:", grade)
15 }
```

编译并运行该程序,输出结果为:

```
默认格式打印:
学号: 100
姓名: 李明
成绩: 91.5
```

通过例 3-2 的输出结果可以看出和例 3-1 的输出效果一模一样,所以在遇到要求按照默认格式输出,还需输出一个空格的场合,通常使用 Println() 函数来完成,而且代码比使用 Print() 函数更简洁清晰。

3. Printf() 函数

前面介绍的 Print() 和 Println() 函数都是采用默认格式输出数据,如果要按照更灵活的格式输出数据就要使用 Printf() 函数,Printf() 函数的原型定义如下:

```
func Printf(format string, a ...interface{}) (n int, err error)
```

fmt 包中的 Printf() 函数类似于 C 语言的 printf() 函数,会按照格式化(format)字符串将操作数格式化输出到标准输出设备上,Printf() 函数也会返回输出的字节数和错误类型。

Go 语言提供的格式化字符串类型非常丰富,其含义和作用如表 3-1 所示。

表 3-1 format 字符串的含义和作用

作用对象	格式字符	说 明
所有类型	%v	以数据对象的基本格式输出
	%#v	输出数值的同时输出 Go 语法表示
	%T	输出数据类型,比如 int、float 等
	%%	输出“%”
布尔型	%t	输出布尔值“true”或“false”
整型	%b	以二进制格式输出
	%c	以 Unicode 字符格式输出
	%d	以十进制格式输出
	%o	以八进制格式输出,0~7
	%q	输出的每个字符自动加单引号
	%x	以十六进制格式输出,使用小写字母 a~f
	%X	以十六进制格式输出,使用大写字母 A~F
	%U	Unicode 格式: U+1234,等价于“U+%04X”
浮点型、复数	%b	无小数部分、两位指数的科学记数法,和 strconv.FormatFloat 的‘b’转换格式一致
	%e	科学记数法,如: -1234.456e+78
	%E	科学记数法,如: -1234.456E+78
	%f	有小数部分,但无指数部分,如: 123.456
	%g	根据实际情况采用 %e 或 %f 格式(以获得更简洁的输出)
	%G	根据实际情况采用 %E 或 %f 格式(以获得更简洁的输出)
字符串、切片	%s	直接输出字符串或切片
	%q	输出字符串的同时自动加双引号
	%x	每个字节用两字符十六进制数表示(使用小写 a~f)
	%X	每个字节用两字符十六进制数表示(使用大写 A~F)
指针	%p	使用以 0x 开头的十六进制数表示
其他格式符	+	输出数值的正负号,对 %q(%+q)按 ASCII 码输出
	-	使用空格填充右侧空缺,而不是默认的左侧
	#	切换格式: 在八进制前加 0(%#o),在十六进制前加 0x(%#x)或 0X(%#X),去掉指针的 0x(%#p)。对于 %q(%#q)输出无修饰符的字符串,对于 %U(%#U)输出可打印的 Unicode 字符
	,	对于数字(%d)会在数字前留一个空格,对于切片或字符串(%x)或(%X)会以十六进制输出
	0	用前置 0 代替空格填补空缺

(1) 通用格式化操作符。通过表 3-1 可以看出,%v、%#v、%T、%%这几种操作符可以作用于任何数据类型。尤其是使用操作符“%v”时,用户不需考虑数据的具体类型就可以正确输出,这种功能对用户输出数据来说非常方便。使用操作符“%#v”能输出 Go 语法表示,可以方便程序检错。使用操作符“%T”能输出数据类型,可以方便程序分析。使用操作符“%%”可以输出百分号“%”,可以用于一些统计结果输出。

例 3-3 通用格式化输出。

```

1 //通用格式化输出
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var id int = 100
9     var name string = "李明"
10    var grade float32 = 91.5
11    fmt.Println("通用格式化输出:")
12    fmt.Printf("%v %v %v\n", id, name, grade)
13    fmt.Printf("#v #v #v\n", id, name, grade)
14    fmt.Printf("%T %T %T\n", id, name, grade)
15    fmt.Printf("60% %\n")
16 }

```

编译并运行该程序,输出结果为:

```

通用格式化输出:
100 李明 91.5
100 "李明" 91.5
int string float32
60%

```

(2) 布尔型数据格式输出。通过 2.2.1 节的知识内容知道,在 Go 语言中布尔型数据的取值是“true”和“false”,这和其他高级程序设计语言不同,比如 C 语言中可以使用“0”代表布尔型的“false”,使用“1”代表布尔型的“true”。在使用 Printf() 函数时,格式化操作符“t”用于输出布尔型数据。

例 3-4 布尔型数据输出。

```

1 //布尔型数据输出
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var yes, no = true, false
9     fmt.Println("布尔型数据输出:")
10    fmt.Printf("%t %t\n", yes, no)
11 }

```

编译并运行该程序,输出结果为:

布尔型数据输出:

```
true false
```

(3) 整型数据格式输出。通过 2.2.2 节的知识内容可以了解到,Go 语言中的整型数据类型非常丰富,比如 int 型、int8 型、int32 型等。另外,整型数据在计算机中的表示形式也多种多样,比如二进制、十进制、八进制、十六进制等。最后,由于计算机字符编码不同,比如 ASCII 编码、Unicode 编码、UTF-8 编码等,整型数据也可以按不同编码字符输出。所以,Go 语言中整型数据的格式化输出方式多种多样,非常灵活,如表 3-1 所示。

例 3-5 整型数据格式化输出。

```
1 //整型数据格式化输出
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var year int = 20013
9     fmt.Println("整型数据格式化输出:")
10    fmt.Printf("二进制格式: %b\n", year)
11    fmt.Printf("十进制格式: %d\n", year)
12    fmt.Printf("八进制格式: %o\n", year)
13    fmt.Printf("十六进制格式(a~f): %x\n", year)
14    fmt.Printf("十六进制格式(A~F): %X\n", year)
15    fmt.Printf("数值对应的 Unicode 编码: %c\n", year)
16    fmt.Printf("Unicode 格式: %U\n", year)
17    fmt.Printf("Unicode 编码使用单引号括起来: %q\n", year)
18 }
```

编译并运行该程序,输出结果为:

整型数据格式化输出:

二进制格式: 100111000101101

十进制格式: 20013

八进制格式: 47055

十六进制格式(a~f): 4e2d

十六进制格式(A~F): 4E2D

数值对应的 Unicode 编码: 中

Unicode 格式: U+4E2D

Unicode 编码使用单引号括起来: '中'

(4) 浮点型和复数格式输出。通过 2.2.3 节的知识内容可以了解到,用户通常在进行浮点型数据运算时采用十进小数制形式,而计算机内部则采用指数形式来存储浮点型数据。所以在进行浮点型数据输出时,可以采用十进制小数形式,也可以采用指数形式(科学记数法)。

对于复数,它是由“实部”和“虚部”两部分组成的,“实部”、“虚部”其实质就是两个浮点型数据,可以使用 real() 和 imag() 函数获取,然后再采用合适的浮点型数据输出格式就可以了。

例 3-6 浮点型数据和复数的格式化输出。

```

1 //浮点型数据和复数的格式化输出
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var f float32 = 123.4567
9     var cp = complex(1.2, 3.4)
10    fmt.Println("浮点型数、复数格式化输出:")
11    fmt.Printf("无小数两位指数科学记数法: %b\n", f)
12    fmt.Printf("科学记数法(小写): %e\n", f)
13    fmt.Printf("科学记数法(大写): %E\n", f)
14    fmt.Printf("根据实际情况采用 %e 或 %f(小写): %g\n", f)
15    fmt.Printf("根据实际情况采用 %E 或 %f(大写): %G\n", f)
16    fmt.Printf("只有小数部分无指数部分: %f\n", f)
17    fmt.Printf("保留 2 位小数: %6.2f\n", f)
18    fmt.Printf("复数 %v 的实部 = %g 虚部 = %g\n", cp, real(cp), imag(cp))
19 }

```

编译并运行该程序,输出结果为:

```

浮点型数、复数格式化输出:
无小数两位指数科学记数法: 16181717p-17
科学记数法(小写): 1.234567e+02
科学记数法(大写): 1.234567E+02
根据实际情况采用 %e 或 %f(小写): 123.4567
根据实际情况采用 %E 或 %f(大写): 123.4567
只有小数部分无指数部分: 123.456703
保留 2 位小数: 123.46
复数(1.2+3.4i)的实部 = 1.2 虚部 = 3.4

```

(5) 字符串和切片格式输出。

例 3-7 字符串和切片的格式化输出。

```

1 //字符串和切片的格式化输出
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var str string = "Go language"
9     fmt.Println("字符串和切片格式化输出:")
10    fmt.Printf("直接输出字符串或切片 %s\n", str)
11    fmt.Printf("自动加双引号: %q\n", str)
12    fmt.Printf("每个字节用两个字符十六进制表示(a~f): %x\n", str)

```

```
13   fmt.Printf("每个字节用两个字符十六进制表示(A~F): %X\n", str)
14 }
```

编译并运行该程序,输出结果为:

字符串和切片格式化输出:

直接输出字符串或切片:Go language

自动加双引号:"Go language"

每个字节用两个字符十六进制表示(a~f):476f206c616e6775616765

每个字节用两个字符十六进制表示(A~F):476F206C616E6775616765

(6) 指针类型格式输出。Go 语言也支持指针类型,通过 2.2.2 节的知识内容可以了解到,uintptr 可以用来保存 32 位或 64 位的指针。同样在调用 Printf() 函数时,可以使用格式化操作符“p”输出指针地址。

例 3-8 格式化输出指针地址。

```
1 //格式化输出指针地址
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var i int = 100
9     var i_pointer *int
10    i_pointer = &i
11    fmt.Println("格式化输出指针地址:")
12    fmt.Printf("输出以 0x 开头的指针地址: %p\n", i_pointer)
13 }
```

编译并运行该程序,输出结果为:

格式化输出指针地址:

输出以 0x 开头的指针地址: 0x10df00e8

(7) 其他类型格式输出。除了前面介绍的一些格式化操作符,Go 语言还提供了一些其他格式化操作符,例如“+”、“-”、“#”、“'”、“0”等,它们的作用可以参见表 3-1 中的说明。

例 3-9 其他类型格式输出。

```
1 //其他类型格式输出
2 package main
3
4 import(
5     "fmt"
6 )
7
```



```

8 func main() {
9     var a int = 97
10    var f float32 = -1.32
11    var p *int
12    var str string = "Golang"
13    p = &a
14    fmt.Println("其他类型格式输出:")
15    fmt.Printf("输出数值的正负号: % +d, % +g\n", a, f)
16    fmt.Printf("ASCII 码格式输出: % +q\n", a)
17    fmt.Printf("切换格式(#o, #x, #X): %d % #o % #x % #X\n", a, a, a, a)
18    fmt.Printf("消除指针地址前的 0X: %p % #p\n", p, p)
19    fmt.Printf("以十六进制输出字符串: %s, % x\n", str, str)
20 }

```

编译并运行该程序,输出结果为:

```

其他类型格式输出:
输出数值的正负号: +97, -1.32
ASCII 码格式输出: 'a'
切换格式(#o, #x, #X): 97 0141 0x61 0X61
消除指针地址前的 0X: 0x10df00e8 10df00e8
以十六进制输出字符串: Golang,47 6f 6c 61 6e 67

```

3.3.2 标准输入函数

数据输入(Data Input)是当程序在运行过程中,将系统外部原始数据通过标准输入设备传输给系统内部,并将这些数据以外部格式转换为系统便于处理的内部格式的过程,其方式与使用的输入设备密切相关(比如键盘)。在 Go 标准库 `fmt` 包中,有三种标准输入函数 `Scan()`、`Scanf()` 和 `Scanln()`。

1. `Scan()` 函数

`Scan()` 函数的功能是从标准输入设备读取数据,并将使用空格分割的连续数据顺序存入到参数里,换行也将被视为空格。`Scan()` 函数调用成功,返回正确读取的参数的数量 `n`; 如果少于要求提供的参数数量,函数返回 `err` 并报告错误原因。`Scan()` 函数的原型定义如下:

```
func Scan(a ...interface{}) (n int, err error)
```

例 3-10 使用 `Scan()` 函数从标准输入设备录入数据。

```

1 //使用 Scan()函数从标准输入设备录入数据
2 package main
3
4 import(
5     "fmt"

```



```
6 )
7 func main() {
8     var a int
9     var f float32
10    var str string
11    fmt.Println("准备录入数据:")
12    fmt.Scan(&a,&f,&str)
13    fmt.Println("输出录入结果:")
14    fmt.Println(a,f,str)
15 }
```

编译并运行该程序,测试过程如下:

准备录入数据:

100 3.14 Golang ✓

输出录入结果:

100 3.14 Golang

2. Scanln() 函数

Scanln()函数的功能是从标准输入设备读取数据,并将使用空格分割的连续数据顺序存入到参数里。Scanln()函数调用成功,返回正确读取的参数的数量 n ; 如果少于要求提供的参数数量,函数返回 err 并报告错误原因。

从上面的说明可以看出,Scanln()函数的功能类似于 Scan()函数,但 Scanln()函数会在读取到换行符时终止录入数据,并且在存入最后一条数据时必须有换行或结束符。Scanln()函数的原型定义如下:

```
func Scanln(a ...interface{}) (n int,err error)
```

例 3-11 使用 Scanln()函数从标准输入设备录入数据。

```
1 //使用 Scanln()函数从标准输入设备录入数据
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var a int
9     var f float32
10    var str string
11    fmt.Println("准备录入数据:")
12    fmt.Scanln(&a,&f,&str)
13    fmt.Println("输出录入结果:")
14    fmt.Println(a,f,str)
15 }
```

对该程序分为以下三种方式进行测试：

(1) 输入一条数据，然后换行回车。

准备录入数据：

100 ✓

输出录入结果：

100 0

(2) 输入两条数据，然后换行回车。

准备录入数据：

100 3.14 ✓

输出录入结果：

100 3.14

(3) 输入三条数据，然后换行回车。

准备录入数据：

100 3,14 Golang ✓

输出录入结果：

100 3,14 Golang

3. Scanf() 函数

Scanf()函数的功能是按照格式化字符从标准输入设备读取数据，并将所读取的数据顺序存入到参数里。Scanf()函数调用成功，返回正确读取的参数的数量 n ；如果少于要求提供的参数数量，函数返回 `err` 并报告错误原因。Scanf()函数的原型定义如下：

```
func Scanf(format string,a ...interface{}) (n int,err error)
```

例 3-12 使用 Scanf() 函数格式化录入数据。

```
1 //使用 Scanf()函数格式化录入数据
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     var a int
9     var f float32
10    var str string
11    fmt.Println("准备录入数据：")
12    fmt.Scanf("%d,%f,%s",&a,&f,&str)
13    fmt.Println("输出录入结果：")
14    fmt.Println(a,f,str)
15 }
```

编译并运行该程序,测试过程如下:

```
准备录入数据:
100,3.14,Golang ✓
输出录入结果:
100 3.14 Golang
```

在调用 `Scanf()` 函数时还需注意的是,上面的 `Scanf("%d,%f,%str",&a,&f,&str)` 语句中的“%d,%f,%str”,表示不同参数之间要使用逗号“,”隔开。在从键盘输入数据时,数据间也要使用逗号“,”隔开。输入参数之间除了使用逗号分隔,还可以使用空格、Tab 键或回车键分隔。

3.4 Strings 包

Go 标准库中的 `Strings` 包,提供了对字符串进行常用操作的基本函数,比如像字符串查找、判断字符串是否相等、字符串匹配等。

3.4.1 字符串查找函数

如果要判断一个字符串中是否包含某个子串,可以使用 `Strings` 包中的 `Contains()`、`ContainsAny()`、`ContainsRune()` 和 `Count()` 函数。

1. `Contains()` 函数

`Contains()` 函数用于查找子串是否在指定的字符串中,如果在返回 `true`,否则返回 `false`。`Contains()` 函数的原型定义如下:

```
func Contains(s, substr string) bool
```

在函数 `Contains()` 中,参数 `s` 表示需要查找的主串;参数 `substr` 表示子串。

2. `ContainsAny()` 函数

`ContainsAny()` 函数用于查找字符串中是否包含子串中某个 Unicode 编码格式的字符。如果包含返回 `true`,否则返回 `false`。`ContainsAny()` 函数的原型定义如下:

```
func ContainsAny(s, chars string) bool
```

在函数 `ContainsAny()` 中,参数 `s` 表示需要查找的主串;参数 `chars` 表示保存的 Unicode 字符集。

3. `ContainsRune()` 函数

`ContainsRune()` 函数用于查找字符串中是否包含某个 `rune` 类型的字符。如果包含返回 `true`,否则返回 `false`。`ContainsRune()` 函数的原型定义如下:


```
func ContainsRune(s string, r rune) bool
```

在函数 `ContainsRune()` 中, 参数 `s` 表示需要查找的主串; 参数 `r` 表示 `rune` 字符。

4. `Count()` 函数

`Count()` 函数用于统计子串在指定字符串中出现的次数, 该函数调用成功后返回统计结果, 函数原型定义如下:

```
func Count(s, sep string) int
```

在函数 `ContainsRune()` 中, 参数 `s` 表示需要查找的主串; 参数 `sep` 表示需要统计的子串。

例 3-13 字符串查找操作。

```
1 //字符串查找操作
2 package main
3
4 import(
5     "fmt"
6     "strings"
7 )
8 func main() {
9     var str string = "Hello, World!"
10    var s1, s2, s3, s4 string = "llo", "go", "ll", "l"
11    var ch1, ch2 rune = 'c', 'd'
12    fmt.Println("查找子串是否在指定的字符串中:")
13    fmt.Printf("%q 在 %q 中? %t\n", s1, str, strings.Contains(str, s1))
14    fmt.Printf("%q 在 %q 中? %t\n", s2, str, strings.Contains(str, s2))
15    fmt.Println("查找字符串中是否含有子串中的字符:")
16    fmt.Printf("%q 中含有 %q 的字符? %t\n", str, s1, strings.ContainsAny(str, s1))
17    fmt.Printf("%q 中含有 %q 的字符? %t\n", str, s2, strings.ContainsAny(str, s2))
18    fmt.Println("查找字符串中是否含有某个字符:")
19    fmt.Printf("%q 中含有字符 %q? %t\n", str, ch1, strings.ContainsRune(str, ch1))
20    fmt.Printf("%q 中含有字符 %q? %t\n", str, ch2, strings.ContainsRune(str, ch2))
21    fmt.Println("统计指定字符串包含子串的个数:")
22    fmt.Printf("%q 中含有 %d 个子串 %q.\n", str, strings.Count(str, s3), s3)
23    fmt.Printf("%q 中含有 %d 个子串 %q.\n", str, strings.Count(str, s4), s4)
24 }
```

编译并运行该程序, 输出结果为:

查找子串是否在指定的字符串中:

"llo"在"Hello, World!"中? true

"go"在"Hello, World!"中? false

查找字符串中是否含有子串中的字符:

"Hello, World!"中含有"llo"的字符? True

"Hello, World!"中含有"go"的字符? True

查找字符串中是否含有某个字符:

```
"Hello,World!"中含有字符'c'? false
"Hello,World!"中含有字符'd'? true
统计指定字符串包含子串的个数:
"Hello,World!"中含有 1 个子串"ll".
"Hello,World!"中含有 3 个子串"l".
```

3.4.2 字符串比较函数

如果想要判断两个字符串是否相等,可以使用 `EqualFold()` 函数,如果相等该函数返回 `true`,否则返回 `false`,该函数原型定义如下:

```
func EqualFold(s,t string) bool
```

在函数 `EqualFold()` 中,参数 `s` 表示需要查找的主串;参数 `t` 表示需要比较的普串。

例 3-14 判断两个字符串是否相等。

```
1 //判断两个字符串是否相等
2 package main
3
4 import(
5     "fmt"
6     "strings"
7 )
8 func main() {
9     var str1,str2,str3 string = "Go","go","lang"
10    fmt.Println("判断两个字符串是否相等: ")
11    fmt.Printf("%q 等于 %q? %t\n",str1,str2,strings.EqualFold(str1,str2))
12    fmt.Printf("%q 等于 %q? %t\n",str1,str3,strings.EqualFold(str1,str3))
13 }
```

编译并运行该程序,输出结果为:

```
判断两个字符串是否相等:
"Go"等于"go"? true
"Go"等于"lang"? false
```

通过分析例 3-14 的运行结果可以发现,`EqualFold()` 函数在对字符串进行比较时会忽略大小写。

3.4.3 字符串位置索引函数

如果要判断一个子串在主串中出现的位置索引,可以使用 `Strings` 包中的 `Index()`、`IndexAny()`、`IndexFunc()`、`IndexRune()`、`LastIndex()`、`LastIndexAny()` 和 `LastIndexFunc()` 函数。

1. Index() 函数

Index() 函数用于判断子串在主串中第一次出现的位置, 如果存在, 返回 int, 对应 sep 出现在 s 中的索引位置; 如果不存在, 则返回 -1。该函数原型定义如下:

```
func Index(s, sep string) int
```

在函数 Index() 中, 参数 s 表示需要查找的主串; 参数 sep 表示需要判断的第一次出现位置的字符串。

2. IndexAny() 函数

IndexAny() 函数用于判断 chars 集中任意一个 Unicode 字符在主串中第一次出现的位置, 如果存在, 返回 int, 对应相关字符出现在 s 中的索引位置; 如果不存在, 则返回 -1。该函数原型定义如下:

```
func IndexAny(s, chars string) int
```

在函数 IndexAny() 中, 参数 s 表示需要查找的主串; 参数 chars 表示保存的 Unicode 字符集。

3. IndexFunc() 函数

IndexFunc() 函数用于判断字符串 s 中的每一个传入函数 f() 的字符, 返回符合函数 f() 的第一个字符的位置。如果符合, 返回该字符在 s 中的位置; 如果都不符合, 则返回 -1。该函数原型定义如下:

```
func IndexFunc(s string, f func(rune) bool) int
```

在函数 IndexFunc() 中, 参数 s 表示需要判断的主串; 参数 f 表示一个函数, 该函数参数是 rune 类型, 返回值是 bool, 如果 rune 符合 f 函数的逻辑那么返回 true, 否则返回 false。

4. IndexRune() 函数

IndexRune() 函数用于判断 rune 类型的字符 r 在字符串 s 中第一次出现的位置, 如果存在, 返回 int, 对应 r 出现在 s 中的索引位置; 如果不存在, 则返回 -1。该函数原型定义如下:

```
func IndexRune(s string, r rune) int
```

在函数 IndexRune() 中, 参数 s 表示需要查找的主串; 参数 r 表示 rune 类型的字符。

5. LastIndex() 函数

LastIndex() 函数用于判断子串在主串中最后一次出现的位置, 如果存在, 返回 int, 对应 sep 出现在 s 中的索引位置; 如果不存在, 则返回 -1。该函数原型定义如下:


```
func LastIndex(s, sep string) int
```

在函数 LastIndex() 中, 参数 s 表示需要查找的主串; 参数 sep 表示需要判断的最后一次出现位置的字符串。

6. LastIndexAny() 函数

LastIndexAny() 函数用于判断 chars 集中任意一个 Unicode 字符在主串中最后一次出现的位置, 如果存在, 返回 int, 对应相关字符出现在 s 中的索引位置; 如果不存在, 则返回 -1。该函数原型定义如下:

```
func LastIndexAny(s, chars string) int
```

在函数 LastIndexAny() 中, 参数 s 表示需要查找的主串; 参数 chars 表示需要判断的最后一次出现位置的 Unicode 字符集。

7. LastIndexFunc() 函数

LastIndexFunc() 函数用于判断传入函数 f() 的字符串中, 是否存在符合 f() 限定条件的字符。如果 s 中存在此种字符, 则返回最后一个字符在 s 中的位置; 如果 s 中不存在此种字符, 则返回 -1。该函数原型定义如下:

```
func LastIndexFunc(s string, f func(rune) bool) int
```

在函数 LastIndexFunc() 中, 参数 s 表示需要判断的主串; 参数 f 表示一个函数, 该函数参数是 rune 类型, 返回值是 bool, 如果 rune 符合 f 函数的逻辑那么返回 true, 否则返回 false。

例 3-15 判断子串在主串中出现的位置。

```
1 //判断子串在主串中出现的位置
2 package main
3
4 import(
5     "fmt"
6     "strings"
7 )
8 func main() {
9     var s, sep, chars string = "Golang", "an", "lang"
10    var r rune = 'a'
11    fmt.Printf("%q 第一次在 %q 中出现的索引是: %d\n", sep, s, strings.Index(s, sep))
12    fmt.Printf("%q 中的字符第一次在 %q 中出现的索引是: %d\n", chars, s, strings.
13    IndexAny(s, chars))
14    fmt.Printf("%q 第一个符合 f() 的字符索引是: %d\n", s, strings.IndexFunc(s, f))
15    fmt.Printf("%q 第一次在 %q 中出现的索引是: %d\n", r, s, strings.IndexRune(s, r))
16    fmt.Printf("%q 最后一次在 %q 中出现的索引是: %d\n", sep, s, strings.LastIndex(s, sep))
```

```

16     fmt.Printf("%q 中的字符最后一次在 %q 中出现的索引是: %d\n", chars, s, strings.
LastIndexAny(s, chars))
17     fmt.Printf("%q 最后一个符合 f() 的字符索引是: %d\n", s, strings.LastIndexFunc(s, f))
18 }
19 func f(a rune) bool {
20     if a > 'k' {
21         return true
22     } else {
23         return false
24     }
25 }

```

编译并运行该程序,输出结果为:

```

"an"第一次在"Golang"中出现的索引是: 3
"lang"中的字符第一次在"Golang"中出现的索引是: 2
"Golang"第一个符合 f() 的字符索引是: 1
'a'第一次在"Golang"中出现的索引是: 3
"an"最后一次在"Golang"中出现的索引是: 3
"lang"中的字符最后一次在"Golang"中出现的索引是: 5
"Golang"最后一个符合 f() 的字符索引是: 4

```

3.4.4 字符串追加和替换函数

如果要向一个字符串后面追加内容或替换字符串的内容,可以使用 `Strings` 包中的 `Repeat()` 函数或 `Replace()` 函数。

1. Repeat() 函数

`Repeat()` 函数用于向字符串中追加内容,该函数返回一个新的字符串,这个新字符串由 `s` 重复 `count` 次构成。该函数原型定义如下:

```
func Repeat(s string, count int) string
```

在函数 `Repeat()` 中,参数 `s` 表示需要重复的字符串;参数 `count` 表示需要重复的次数。

2. Replace() 函数

`Replace()` 函数用于把主串 `s` 中的 `old` 子串替换为 `new` 子串,替换次数为 `n` 次。函数调用成功后返回处理后的新串,该函数原型定义如下:

```
func Replace(s, old, new string, n int) string
```

在函数 `Replace()` 中,参数 `s` 表示需要替换的主串;参数 `old` 表示需要替换的子串;参数 `new` 表示用于替换的新子串;参数 `n` 表示需要替换的次数,如果 `n < 0`,则全部替换。

例 3-16 字符串追加和替换。

```
1 //字符串追加和替换
2 package main
3
4 import(
5     "fmt"
6     "strings"
7 )
8 func main() {
9     var s = "na"
10    var count int = 2
11    //字符串追加
12    fmt.Println("ba" + strings.Repeat(s, count))
13    //字符串替换
14    fmt.Println(strings.Replace("google", "o", "oo", 1))
15    fmt.Println(strings.Replace("google", "o", "oo", -1))
16 }
```

编译并运行该程序,输出结果为:

```
banana
goooogle
goooogle
```

3.5 Strconv 包

Go 标准库中的 Strconv 包,提供了字符串与基本数据类型相互转换的一些基本函数。Format 系列函数用于将基本数据类型转换为字符串; Parse 系列函数用于将字符串转换为基本数据类型; Atoi() 和 Itoa() 则是相关函数的简便封装版本。

3.5.1 数值转换为字符串函数

如果要将基本数据类型转换为字符串格式,可以使用 Strconv 包中的 FormatBool()、FormatFloat()、FormatInt() 和 FormatUint() 函数。

1. FormatBool() 函数

FormatBool() 函数用于将布尔型数据转换为字符串形式,该函数原型定义如下:

```
func FormatBool(b bool) string
```

函数 FormatBool() 的参数 b 表示需要被转换的布尔数,可以是“true”或“false”。

2. FormatFloat() 函数

FormatFloat() 函数用于将浮点型数转换为字符串形式,该函数能按照指定格式将浮点数转换成字符串,函数原型定义如下:


```
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
```

在函数 `FormatFloat()` 中, 参数 `f` 表示需要转换的浮点数; 参数 `fmt` 表示浮点数的格式, 可以是 `b`、`e`、`E`、`f`、`g` 或 `G`, 具体含义可以参见表 3-1。参数 `prec` 表示 `e`、`E`、`f`、`g`、`G` 格式浮点数的输出精度, 如果为 `e`、`E`、`f` 格式, `prec` 表示小数点后的有效位数。如果为 `g`、`G` 格式, `prec` 表示全部有效位数。如果 `prec` 为 `-1`, 则以最少有效位数表示该数; 参数 `bitSize` 表示浮点型数的位数, 比如 32 或 64, 分别对应 `float32` 和 `float64`。

3. FormatInt() 函数

`FormatInt()` 函数用于将整型数转换为字符串形式, 该函数能按照指定格式将整型数转换成字符串, 函数原型定义如下:

```
func FormatInt(i int64, base int) string
```

在函数 `FormatFloat()` 中, 参数 `i` 表示需要转换的整数; 参数 `base` 表示整型数的进制, 比如 2、8、10、16, 分别对应二进制、八进制、十进制和十六进制。

4. FormatUint() 函数

`FormatUint()` 函数和 `FormatInt()` 函数的功能基本相似, 只不过是将其无符号整数转换为字符串形式。该函数原型定义如下:

```
func FormatUint(i uint64, base int) string
```

在函数 `FormatUint()` 中, 参数 `i` 表示需要转换的无符号整数; 参数 `base` 表示整型数的进制, 比如 2、8、10、16, 分别对应二进制、八进制、十进制和十六进制。

例 3-17 将数值转换成字符串。

```
1 //将数值转换成字符串
2 package main
3
4 import(
5     "fmt"
6     "strconv"
7 )
8 func main() {
9     var b bool = false
10    var f float64 = 3.14
11    var i int64 = -1024
12    var ui uint64 = 1024
13    fmt.Printf("将布尔数 %t 转换成字符串: %q\n", b, strconv.FormatBool(b))
14    fmt.Printf("将浮点数 %f 转换成字符串: %q\n", f, strconv.FormatFloat(f, 'f', 2, 32))
15    fmt.Printf("将整数 %d 转换成字符串: %q\n", i, strconv.FormatInt(i, 10))
16    fmt.Printf("将无符号整数 %d 转换成字符串: %q\n", ui, strconv.FormatUint(ui, 10))
17 }
```

编译并运行该程序,输出结果为:

```
将布尔数 false 转换成字符串: "false"  
将浮点数 3.140000 转换成字符串: "3.14"  
将整数 -1024 转换成字符串: "-1024"  
将无符号整数 1024 转换成字符串: "1024"
```

3.5.2 字符串转换为数值函数

如果要将字符串转换为基本数据类型,可以使用 `Strconv` 包中的 `ParseBool()`、`ParseFloat()`、`ParseInt()` 和 `ParseUint()` 函数。

1. `ParseBool()` 函数

`ParseBool()` 函数用于将字符串转换成布尔型数据,该函数调用成功,返回对应的布尔数 `value`; 否则,返回一个错误类型 `err`。该函数的原型定义如下:

```
func ParseBool(str string) (value bool, err error)
```

在函数 `ParseBool()` 中,参数 `str` 表示需要转换的布尔数的字符串形式。

2. `ParseFloat()` 函数

`ParseFloat()` 函数用于将字符串转换成指定格式的浮点数,该函数调用成功,返回对应的浮点数 `f`; 否则,返回一个错误类型 `err`。该函数的原型定义如下:

```
func ParseFloat(s string, bitSize int) (f float64, err error)
```

在函数 `ParseFloat()` 中,参数 `s` 表示需要转换的浮点数的字符串形式; 参数 `bitSize` 表示浮点型数的位数,比如 32 或 64,分别对应 `float32` 和 `float64`。

3. `ParseInt()` 函数

`ParseInt()` 函数用于将字符串转换为参数指定的整型数,该函数调用成功,返回对应的整型数 `i`; 否则,返回一个错误类型 `err`。该函数原型定义如下:

```
func ParseInt(s string, base int, bitSize int) (i int64, err error)
```

在函数 `ParseInt()` 的参数中,`s` 表示需要转换的整型数的字符串形式; 参数 `base` 表示整型数的进制,比如 2、8、10、16,分别对应二进制、八进制、十进制和十六进制; 参数 `bitSize` 表示返回结果的位数,比如 0、8、16、32 和 64,分别对应 `int`、`int8`、`int16`、`int32` 和 `int64`。

4. `ParseUint()` 函数

函数 `ParseUint()` 和函数 `ParseInt()` 的功能基本类似,只不过它是将字符串转换为参数指定的无符号整数,该函数调用成功,返回对应的无符号整数 `n`; 否则,返回一个错误类型

err。该函数原型定义如下：

```
func ParseUint(s string, b int, bitSize int) (n uint64, err error)
```

在函数 ParseUint() 中, 参数 s 表示需要转换的无符号整数的字符串形式; 参数 base 表示整型数的进制, 比如 2、8、10、16, 分别对应二进制、八进制、十进制和十六进制; 参数 bitSize 表示返回结果的位数, 比如 0、8、16、32 和 64, 分别对应 uint、uint8、uint16、uint32 和 uint64。

例 3-18 将字符串转换成数值。

```
1 //将字符串转换成数值
2 package main
3
4 import(
5     "fmt"
6     "strconv"
7 )
8 func main() {
9     var strb, strf, stri, strui string = "false", "3.14", "-1024", "1024"
10    var b bool
11    var f float64
12    var i int64
13    var ui uint64
14    b, _ = strconv.ParseBool(strb)
15    f, _ = strconv.ParseFloat(strf, 32)
16    i, _ = strconv.ParseInt(stri, 10, 32)
17    ui, _ = strconv.ParseUint(strui, 10, 32)
18    fmt.Printf("将字符串 %q 转换成布尔数: %t\n", strb, b)
19    fmt.Printf("将字符串 %q 转换成浮点数: %f\n", strf, f)
20    fmt.Printf("将字符串 %q 转换成整数: %d\n", stri, i)
21    fmt.Printf("将字符串 %q 转换成无符号整数: %d\n", strui, ui)
22 }
```

编译并运行该程序, 输出结果为:

```
将字符串"false"转换成布尔数: false
将字符串"3.14"转换成浮点数: 3.140000
将字符串"-1024"转换成整数: -1024
将字符串"1024"转换成无符号整数: 1024
```

3.5.3 Atoi() 和 Itoa() 函数

除了 FormatInt() 和 ParseInt() 函数, Strconv 包中还提供了更简单的 Atoi() 和 Itoa() 函数, 用于整型数和字符串之间的相互转换。

1. Atoi() 函数

函数 Atoi() 的作用和函数 ParseInt() 一样, 是将字符串转换为整数, 只不过更简单。该函数调用成功, 返回对应的整型数 i; 否则, 返回一个错误类型 err。该函数原型定义如下:


```
func Atoi(s string) (i int, err error)
```

在函数 Atoi() 中, 参数 s 表示需要转换的整数的字符串形式。

2. Itoa() 函数

函数 Itoa() 和函数 FormatInt() 的作用一样, 是将整数转化成字符串形式。该函数调用成功, 返回整数的字符串形式。

```
func Itoa(i int) string
```

在函数 Itoa() 中, 参数 i 是需要转换的整数。

例 3-19 整数和字符串的相互转换。

```
1 //整数和字符串的相互转换
2 package main
3
4 import(
5     "fmt"
6     "strconv"
7 )
8 func main() {
9     var i int = -1024
10    var s string = "1024"
11    value, _ := strconv.Atoi(s)
12    str := strconv.Itoa(i)
13    fmt.Printf("将字符串 %q 转换成整数: %d\n", s, value)
14    fmt.Printf("将整数 %d 转换成字符串: %q\n", i, str)
15 }
```

编译并运行该程序, 输出结果为:

将字符串 "1024" 转换成整数: 1024

将整数 -1024 转换成字符串: "-1024"

3.6 顺序结构程序举例

通过第 2 章的学习掌握了程序中常量与变量的声明方法, 通过本章的学习掌握了数据的输入、处理和输出等方法, 这样就可以利用这些知识设计简单的顺序结构程序了。下面介绍几个顺序程序设计的例子, 对前面所学知识加以验证和巩固。

3.6.1 求平均值

例 3-20 从键盘输入三个整型数, 然后计算它们的平均值, 最后输出计算结果, 小数点后保留两位有效位。

从这个程序的任务需求可以看出,完成这个计算任务共需 3 步:

① 数据输入; ② 计算平均值; ③ 输出计算结果。

由于 Go 语言是强类型语言,所以在设计本例算法时还需注意数据类型一致问题,题目中要求输入的三个数据为整型,而它们的平均数应该为浮点型,所以在计算平均数时要进行类型转换,即将整型转换为浮点型。程序代码如下:

```
1 //计算平均数
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var sum1,sum2,sum3 int
10    var average float32
11    fmt.Println("请输入 3 个整数: ")
12    fmt.Scanf("%d, %d, %d",&sum1,&sum2,&sum3)
13    fmt.Println("计算平均数...")
14    average = float32(sum1 + sum2 + sum3) / 3.0
15    fmt.Printf("输出计算结果: %6.2f\n",average)
16 }
```

编译并运行该程序,测试过程如下:

```
请输入 3 个整数:
13,22,-7 ✓
计算平均数...
输出计算结果: 9.33
```

3.6.2 计算三角形面积周长

例 3-21 从键盘输入三角形三个边长,求三角形面积和周长,输出计算结果,小数点后保留两位有效位。

假设三角形的三边分别为 a、b、c,周长为 l,面积为 area,则

$$l = a + b + c$$

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)} \quad (\text{其中 } s = l/2)$$

所以,解决该问题至少需要以下几步:

(1) 输入三角形三边 a、b、c。

(2) 计算三角形周长 l。

(3) 计算半周长 hl。

(4) 计算三角形面积 area。

(5) 输出计算结果。

该程序需要使用 math 包中的 Sqrt() 函数计算平方根,所以在使用前要导入 math 包。

程序代码如下：

```
1 //计算三角形面积和周长
2 package main
3
4 import(
5     "fmt"
6     "math"
7 )
8
9 func main() {
10     var a,b,c,l,hl,area float64
11     fmt.Println("请输入三角形三边: ")
12     fmt.Scanf("%f, %f, %f",&a,&b,&c)
13     l = a + b + c
14     fmt.Printf("三角形周长 = %6.2f\n",l)
15     hl = l * 0.5
16     area = math.Sqrt(hl * (hl - a) * (hl - b) * (hl - c))
17     fmt.Printf("三角形面积 = %6.2f\n",area)
18 }
```

编译并运行该程序,测试过程如下:

请输入三角形三边:

5.1,4.5,3.2 ✓

三角形周长 = 12.80

三角形面积 = 7.11

3.6.3 求解一元二次方程

例 3-22 求解一元二次方程 $ax^2+bx+c=0$ ($b^2-4ac>0$) 的实数根,输出计算结果,小数点后保留两位有效位。

由一元二次方程的求根公式:

$$x = (-b \pm \sqrt{b^2 - 4ac}) / 2a$$

设两个中间量 $p, q, p = -b/2a, q = \sqrt{b^2 - 4ac}/2a$,

可得 $x_1 = p + q, x_2 = p - q$

所以,解决该问题需要以下几个步骤:

(1) 输入一元二次方程的三个系数。

(2) 计算判别式 $disc$ 。

(3) 计算中间量 p, q 。

(4) 计算方程的根 x_1, x_2 。

(5) 输出方程的根 x_1, x_2 。

例 3-22 的代码如下:

```
1 //计算一元二次方程的根
2 package main
3
```



```

4 import(
5     "fmt"
6     "math"
7 )
8
9 func main() {
10     var a,b,c,disc,x1,x2,p,q float64
11     fmt.Println("请输入一元二次方程三个系数:")
12     fmt.Scanf("%f, %f, %f", &a, &b, &c)
13     disc = b * b - 4 * a * c
14     p = -b / (2 * a)
15     q = math.Sqrt(disc) / (2 * a)
16     x1 = p + q
17     x2 = p - q
18     fmt.Printf("一元二次方程的根 x1 = %6.2f x2 = %6.2f\n", x1, x2)
19 }

```

编译并运行该程序,测试过程如下:

请输入一元二次方程三个系数:

1.1,3.1,2.1 ✓

一元二次方程的根 x1 = -1.13 x2 = -1.69

小结

本章主要介绍了 Go 语言的顺序结构程序设计方法,并介绍了 Go 顺序程序结构、简单语句和复合语句。另外,还介绍了在设计 Go 程序时的注意事项,包括语句单位划分、左大括号约定、注释语句等。最后介绍了 fmt 包中的输入输出函数,strings 包中的字符串处理函数,strconv 包中的字符串转换函数。

通过这一章的学习,首先要掌握顺序结构程序设计的方法,然后要掌握程序中的数据输入输出方法,最后要灵活运用 Go 标准库函数,设计出高效简洁的顺序结构程序。

习题

3.1 输入一个三位整数,求出该数每个位上的数字之和。如 123,每个位上的数字和就是 $1+2+3=6$ 。

3.2 输入三个 float64 类型浮点数,分别求出它们的和、平均值、平方和以及平方和的开方,并输出所求出的各个值。

3.3 设 f 表示华氏温度、 c 表示摄氏温度、 k 表示绝对温度,将华氏温度转换为摄氏温度和绝对温度的公式分别为:

$$c = 5/9 \times (f - 32) \text{ (摄氏温度)}$$

$$k = 273.16 + c \text{ (华氏温度)}$$

编写程序,要求通过键盘输入 f 的值,计算 c 和 k 的值并输出。

3.4 编写程序,把极坐标 (r, θ) 转换为直角坐标 (x, y) ,其中 θ 的单位为度。转换公式是:

$$x = r \times \cos \theta$$

$$y = r \times \sin \theta$$

提示:要调用 $\cos()$ 函数和 $\sin()$ 函数需导入 `math` 包。

3.5 通过键盘输入英文字符串,并统计字符串中的英文字母的个数,同时输出字符串字节长度。

例 4-2 1.4

例 4-2 if 语句的使用,条件表达式为关系表达式或逻辑表达式

在 Go 语言中,if 语句用于根据条件执行不同的代码块。其基本语法如下:

1.1.4 if 语句的语法

在 Go 语言中,if 语句用于根据条件执行不同的代码块。其基本语法如下:

在 Go 语言中,if 语句用于根据条件执行不同的代码块。其基本语法如下:

在 Go 语言中,if 语句用于根据条件执行不同的代码块。其基本语法如下:

在 Go 语言中,if 语句用于根据条件执行不同的代码块。其基本语法如下:

在 Go 语言中,if 语句用于根据条件执行不同的代码块。其基本语法如下:

在 Go 语言中,if 语句用于根据条件执行不同的代码块。其基本语法如下:

在 Go 语言中,if 语句用于根据条件执行不同的代码块。其基本语法如下:

第4章

Go 选择结构程序设计

在设计应用程序时,程序语句不仅仅是顺序地、一条接一条地执行,往往需要依据某种条件,让程序转向不同的流程去执行,这就是选择结构(Selective Structure)。在选择结构程序中,通常要使用条件表达式来描述外部条件。Go 语言提供了三种语句来实现选择结构:if 语句、switch 语句和 select 语句。

4.1 if 语句

if 语句也称为条件选择语句,它是使用 if 语句来判定所给定的条件是否满足,根据判定的结果(true 或 false)决定执行给出的两种操作之一。

4.1.1 if 语句的形式

在 Go 语言中,共提供了三种形式的 if 语句:if 语句、if else 语句和 if else if 语句。

1. 单分支选择结构

如果一个条件选择程序中仅有一个条件分支,则称该结构为单分支选择结构(Single-branch Selection Structure)。单分支选择结构使用 if 语句实现即可,一般形式如下:

```
if 表达式{  
    语句 1  
}
```

在单分支 if 语句中,表达式可以是布尔值、关系表达式或逻辑表达式,if 会检测表达式的值,如果表达式的值为 true,则执行语句 1;如果表达式的值为 false,则什么也不执行。语句 1 可以是一条语句,也可以是复合语句或任何一种控制语句。

例 4-1 if 语句的使用,条件表达式为 boolean 值。

```
1 //if 语句的使用,条件表达式为 boolean 值。  
2 package main  
3  
4 import(  
5     "fmt"  
6 )
```



```
7
8 func main() {
9     //条件为 true, 执行{}中的语句.
10    if true {
11        fmt.Println("The condition is true!")
12    }
13    //条件为 false, 跳过{}执行后续语句.
14    if false {
15        fmt.Println("The condition is false!")
16    }
17    //if 语句外的后续语句.
18    fmt.Println("OVER")
19 }
```

编译并运行该程序, 输出结果为:

```
The condition is true!
OVER
```

例 4-2 if 语句的使用, 条件表达式为关系表达式或逻辑表达式。

```
1 //if 语句的使用, 关系表达式或逻辑表达式的值作为判断条件.
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var a,b int
10    //通过 Scanln() 函数, 从键盘输入 a,b 的值.
11    fmt.Scanln(&a,&b)
12    //关系表达式的值作为判断条件.
13    if a > b {
14        fmt.Println("a > b")
15    }
16    //逻辑表达式的值作为判断条件.
17    if (a != 0) && (a > 0) {
18        fmt.Println("a is positive number")
19    }
20    if (a != 0) && (a < 0) {
21        fmt.Println("a is negative number")
22    }
23    //if 语句外的后续语句.
24    fmt.Println("OVER")
25 }
```

编译并运行该程序, 测试过程如下:

```

从键盘输入: 5 3 ✓
输出结果为: a > b
             a is positive number
             OVER

```

2. 双分支选择结构

如果一个条件选择程序中有两个条件分支,则称该结构为双分支选择结构(Dual-branch Selection Structure)。双分支选择结构可以使用 if else 语句来实现,一般形式如下:

```

if 表达式{
    语句 1
} else{
    语句 2
}

```

在双分支选择 if else 语句中,如果表达式的值为 true,则执行语句 1; 如果表达式的值为 false,则执行语句 2,对表达式和语句的规定和 if 语句一样。

例 4-3 if else 语句的使用,找出最大数。

```

1 //使用 if else 语句找出最大数
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var a,b int
10    //通过 Scanln()函数,从键盘输入 a,b 的值.
11    fmt.Scanln(&a,&b)
12    if a > b {
13        //条件为 true 执行这里
14        fmt.Printf("MAX = %d\n",a)
15    }else {
16        //条件为 false 执行这里
17        fmt.Printf("MAX = %d\n",b)
18    }
19}

```

编译并运行该程序,测试过程如下:

```

从键盘输入: 56 127 ✓
输出结果为: MAX = 127

```


3. 多分支选择结构

如果一个条件选择程序中有多个条件分支,则称该结构为多分支选择结构(Multi-branch Selection Structure)。多分支选择结构可以使用 if else if 语句来实现,一般形式如下:

```
if 表达式 1{
    语句 1
} else if 表达式 2{
    语句 2
} else if 表达式 3{
    语句 3
    :
} else if 表达式 n{
    语句 n
} else {
    语句 n+1
}
```

在多条件分支 if else if 语句中有多个条件表达式,所以它可以控制程序进行多条件选择执行。比如不符合条件 1,可以继续检测是否符合条件 2,依此类推。如果所有条件都不符合,最后执行 else 里的语句。

例 4-4 if else if 多条件判断语句的使用,打印成绩等级。

```
1 //使用多条件判断打印成绩等级
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var grade int
10    //通过 Scanln() 函数,从键盘输入成绩 grade.
11    fmt.Scanln(&grade)
12    if (grade >= 90) && (grade <= 100) {
13        //成绩在 90~100 区间等级为 A
14        fmt.Println("A")
15    } else if (grade >= 80) && (grade < 90) {
16        //成绩在 80~89 区间等级为 B
17        fmt.Println("B")
18    } else if (grade >= 70) && (grade < 80) {
19        //成绩在 70~79 区间等级为 C
20        fmt.Println("C")
21    } else {
22        //成绩在 70 以下等级为 D
23        fmt.Println("D")
24    }
25 }
```


编译并运行该程序,测试过程如下:

从键盘输入: 87 ✓

输出结果为: B

4.1.2 if 语句的嵌套

在 if 语句中又包含一个或多个 if 语句的用法,称为 if 语句的嵌套(Nested)。if 语句的嵌套使用形式如下:

```
if 表达式{  
    if 表达式{  
        语句 1  
    }else{  
        语句 2  
    }  
}else{  
    if 表达式{  
        语句 1  
    }else{  
        语句 2  
    }  
}
```

→ 内嵌 if 语句

→ 内嵌 if 语句

在使用 if 嵌套语句的时候应当注意,else 总是与它上面最近的未配对的 if 配对。下面的例子是使用 if 嵌套语句判断闰年。

例 4-5 使用 if 嵌套语句判断闰年。

```
1 //使用 if 嵌套语句判断闰年  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7  
8 func main() {  
9     var year int  
10    //通过 Scanln()函数,从键盘输入 year 的值。  
11    fmt.Scanln(&year)  
12    if year % 4 == 0 {  
13        if year % 100 == 0 {  
14            if year % 400 == 0 {  
15                //能被 4 整除,且能被 100 整除,且能被 400 整除是闰年。  
16                fmt.Printf("%d is leap year.\n", year)  
17            } else {  
18                //能被 4 整除,且能被 100 整除,但不能被 400 整除不是闰年。  
19                fmt.Printf("%d is not leap year.\n", year)  
20            }  
21        }  
22    }  
23 }
```

```
20     }
21     } else {
22         //能被 4 整除,且不能被 100 整除是闰年.
23         fmt.Printf("%d is leap year.\n", year)
24     }
25 } else
26     //不能被 4 整除不是闰年.
27     fmt.Printf("%d is not leap year.\n", year)
28 }
29 }
```

编译并运行该程序,测试过程如下:

从键盘输入: 2013 ✓

输出结果为: 2013 is not leap year.

4.1.3 if 语句的注意事项

每一种程序设计语言几乎都有 if 语句,但 Go 语言中的 if 语句有一些自身的特点,在使用时一定要注意。

(1) 在 Go 语言中,左大括号“{”必须和 if 语句放在同一行,否则编译会出错。

例如,下面的语句“{”使用正确:

```
if a > 0{
}
```

下面的语句“{”使用错误:

```
if a > 0
{
}
```

(2) 在其他语言中,if 语句中的条件表达式通常都会使用括号括起来,但在 Go 语言中,条件表达式没有括号。Go 语言中,条件表达式和 if 关键字之间使用空格隔开即可。

例如,判断 a 是否大于 0,正确的 if 语句是:

```
if a > 0 {
}
```

下面的语句是错误的:

```
if (a > 0){
}
```

(3) Go 语言的 if 语句还支持初始化条件表达式,如 `if a := 1; a > 0`。在这条语句中,先给条件变量 a 赋初值,然后再比较。初始化语句和条件语句之间使用“;”隔开。

这里要注意,条件变量 `a` 的作用范围仅在 `if` 语句块中,当 `if` 语句执行完后,条件变量 `a` 就不存在了,相当于条件变量 `a` 是 `if` 语句中的一个局部变量。

比如下面的程序,试图在 `if` 语句块之外打印条件变量 `a` 的值,编译时编译器会报错,提示变量 `a` 未定义(undefined)。

```
if a := 1; a > 0 {
}
fmt.Println(a)
```

如果在 `if` 语句块外部也定义了一个同名的变量 `a`,那么在 `if` 语句块中,外部变量 `a` 就会被内部条件变量 `a` 覆盖、隐藏起来,整个 `if` 语句块中用到的都是条件变量 `a` 的值。调试并运行例 4-6,观察输出结果。

例 4-6 在 `if` 语句中初始化条件表达式。

```
1 //if 语句初始化条件表达式
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     a := 10
10    if a := 1; a > 0 {
11        //打印 if 语句块中条件变量 a 的值
12        fmt.Println(a)
13    }
14    //打印 if 语句块外变量 a 的值
15    fmt.Println(a)
16 }
```

编译并运行该程序,输出结果为:

```
1
10
```

(4) 在有返回值的函数中,不允许在 `if` 语句块中执行 `return` 语句,否则编译会失败。例如,下面的使用方式:

```
func example(x int) int {
    if x == 0 {
        return 10
    } else {
        return x
    }
}
```

上述语句在编译时,编译器会报错,错误信息为:

function ends without a return statement

产生失败的原因是 Go 编译器无法找到终止该函数的 return 语句。

4.2 switch 语句

if 语句一般只有两个分支可供选择,而实际问题中常常需要用到多分支选择,例如数据统计分类。当然多分支选择结构也可以使用 if 嵌套语句来处理,但如果分支较多,则嵌套的 if 语句层次就较多,程序代码冗长且可读性降低。Go 语言提供了 switch 语句,可以很简便地实现多条件选择结构。

4.2.1 switch 语句结构

switch 语句是另一种多分支选择控制语句,其特点是可以根据一个表达式的多种值,选择多个 case 分支,因而也被称为“开关”语句。case 分支语句依据控制表达式的值,选择执行相关的程序语句。switch 语句的一般形式如下:

```
switch 条件表达式 {  
    case 常量表达式 1:  
        语句 1  
    case 常量表达式 2:  
        语句 2  
        :  
    case 常量表达式 n:  
        语句 n  
    default:  
        语句 n+1  
}
```

switch 语句的控制类型由 switch 条件表达式决定,控制类型可以是任意 Go 语言所支持的数据类型。每一个 switch 分支常量表达式值的类型,必须和控制类型保持一致,如果不一致,必须显式地转换成同一种类型。

如果同一个 switch 语句中,有两个或两个以上的 switch 分支的常量表达式取得相同的值,则编译时会出错。另外,每一个 switch 语句最多只能有一个 default 分支。

switch 语句的执行步骤如下:

- (1) 计算出 switch 表达式的值,并转换成控制类型。
- (2) 如果 switch 条件表达式的值等于某个 switch 分支的常量表达式的值,则程序控制跳转到这个 case 标号后的语句列表中执行。
- (3) 如果 switch 条件表达式的值,无法与 switch 语句中任何一个 case 常量表达式的值相匹配,而且 switch 语句中有 default 分支,则程序会控制跳转到 default 标号后的语句列表中执行。
- (4) 如果 switch 条件表达式的值,无法与 switch 语句中任何一个 case 常量表达式的值

相匹配,且 switch 语句中没有 default 分支,则程序控制会跳转到 switch 语句的结尾,switch 语句执行结束。

例 4-7 使用 switch 语句实现简单计算器。

```
1 //switch 语句实现简单计算器
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     //操作数 a,b
10    var a,b int
11    //算术运算符 + - * /
12    var op byte
13    //通过 Scanf()函数,格式化从键盘输入操作数和运算符.
14    fmt.Scanf("%d %c %d",&a,&op,&b)
15    switch op {
16    case '+':
17        fmt.Println(a+b) //加法
18    case '-':
19        fmt.Println(a-b) //减法
20    case '*':
21        fmt.Println(a*b) //乘法
22    default:
23        fmt.Println(a/b) //除法
24    }
25 }
```

编译并运行该程序,测试过程如下:

从键盘输入: 12 * 6 ✓
输出结果为: 72

4.2.2 switch 语句的特殊形式

switch 语句除了基本使用方式,还有三种特殊使用方式: 多选项 case 语句、继续执行 case 语句和无条件表达式 switch 语句。

1. 多选项 case 语句

有时在一条 case 语句中可以对多个条件值进行测试,任意一个条件满足都会执行 case 语句体,形式如下:

```
switch 条件表达式{
case 常量表达式 1:
```

```
    语句 1
case 常量表达式 2, 常量表达式 3:
    语句 2
    :
case 常量表达式 n:
    语句 n
default:
    语句 n+1
}
```

在以上形式中,第二条 case 语句有两个常量表达式:常量表达式 2 和常量表达式 3,这两个常量表达式之间使用“,”隔开。Case 语句将对这两个常量表达式都进行测试,任何一个常量表达式满足要求,都会执行语句 2。

例 4-8 多选项 case 语句的使用,从键盘输入编程语言的名字作为搜索关键字,switch 语句找到和该关键字相符的分支,打印相关信息。比如 c 和 C 都表示 C 语言,go 和 golang 都表示 Go 语言,多选项 case 语句满足这种设计要求。

```
1 //多选项 case 语句
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     //搜索字符串变量
10    var searchstr string
11    //通过 Scanf()函数,格式化从键盘输入搜索字符串.
12    fmt.Scanf("% s",&searchstr)
13    switch searchstr {
14    case "c","C":
15        //搜索字符串是 c 或 C 都执行下面语句
16        fmt.Println("C programing language")
17    case "go","golang":
18        //搜索字符串是 go 或 golang 都执行下面语句
19        fmt.Println("Go programing language")
20    case "java":
21        //搜索字符串是 java 执行下面语句
22        fmt.Println("Java programing language")
23    default:
24        //没有搜索到
25        fmt.Println("Not find the result!")
26    }
27 }
```

编译并运行该程序,测试过程如下:

从键盘输入: golang ✓
输出结果为: Go programing language

2. fallthrough 语句

通常情况下,switch 语句检测到第一个符合条件的 case 分支,就去执行该 case 分支程序,执行完后自动退出 switch 语句。有时希望在执行完一个 case 分支后,不退出 switch 语句,继续执行下一个 case 分支。这时,可以显式地调用 fallthrough 语句,告诉 switch 语句希望继续执行下一个 case 分支而不退出。fallthrough 语句的调用形式如下:

```
switch 条件表达式{
case 常量表达式 1:
    语句 1
    fallthrough
case 常量表达式 2:
    语句 2
    :
case 常量表达式 n:
    语句 n
default:
    语句 n+1
}
```

在以上形式中,第一条 case 语句中执行了 fallthrough 语句,也就意味着如果第一个分支能够执行,那它执行完后不马上跳出 switch 语句,而是接着执行第二个分支语句。

例 4-9 fallthrough 语句的使用。

```
1 //fallthrough 语句的使用
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var i int
10    //通过 Scanf()函数,格式化从键盘输入变量 i 的值.
11    fmt.Scanf("% d",&i)
12    switch i {
13    case 0:
14        fmt.Println("0")
15    case 1:
16        fmt.Println("1")
17    case 2:
18        fmt.Println("2")
19        fallthrough           //继续执行下一分支
20    case 3:
21        fmt.Println("3")
22    default:
23        fmt.Println("4")
24    }
25 }
```

编译并运行例 4-9,测试结果如下:

```
从键盘输入: 0 ✓  
输出结果为: 0  
从键盘输入: 1 ✓  
输出结果为: 1  
从键盘输入: 2 ✓  
输出结果为: 2  
3  
从键盘输入: 3 ✓  
输出结果为: 3
```

因为在第二个分支 case 语句中调用了 fallthrough 语句,所以当第二个分支能够执行时,第三个程序分支也会被执行。

3. 无条件表达式 switch 语句

如果 switch 关键字后面没有条件表达式,则必须在 case 语句中进行条件判断,这种形式的 switch 语句,其作用类似于 if else if 语句。形式如下:

```
switch {  
case 条件表达式 1:  
    语句 1  
case 条件表达式 2:  
    语句 2  
    :  
case 条件表达式 n:  
    语句 n  
default:  
    语句 n + 1  
}
```

例 4-10 使用无条件表达式 switch 语句,实现例 4-4 打印成绩等级。

```
1 //使用无条件表达式 switch 语句打印成绩等级  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7  
8 func main() {  
9     var grade int  
10    //通过 Scanln() 函数,从键盘输入成绩 grade.  
11    fmt.Scanln(&grade)  
12    switch {  
13    case (grade >= 90) && (grade <= 100):  
14        //成绩在 90~100 区间等级为 A  
15        fmt.Println("A")  
16    case (grade >= 80) && (grade < 90):  
17        //成绩在 80~89 区间等级为 B
```

```
18     fmt.Println("B")
19     case (grade >= 70) && (grade < 80):
20         //成绩在 70~79 区间等级为 C
21         fmt.Println("C")
22     default:
23         //成绩在 70 以下等级为 D
24         fmt.Println("D")
25 }
26 }
```

编译并运行该程序,测试过程如下:

从键盘输入: 96 ✓
输出结果为: A

4.2.3 switch 语句的注意事项

通过前面的知识了解到,switch 语句可以非常灵活地实现多条件分支控制结构,但在使用 switch 语句时,还有几个事项需要注意。

(1) 左大括号“{”必须和 switch 语句同处一行,否则编译会出错。

例如,下面的语句“{”使用正确:

```
switch grade{
}
```

下面的语句“{”使用错误:

```
switch grade
{
}
```

(2) 条件表达式不限制为常量或者整数。

(3) 不需要使用 break 语句跳出 case 语句。

(4) 和 if 语句一样,switch 语句也支持初始化条件表达式。

4.3 选择结构程序举例

本节将列举两个日常生活、学习中的常见事例:解方程和打印日期信息,分别使用 if 语句和 switch 语句进行问题求解。

4.3.1 解一元二次方程

例 4-11 使用 if 语句求解一元二次方程 $ax^2+bx+c=0$ 的根。

在例 3-22 中已经介绍过一元二次方程式的求根算法,只不过前提条件是一种理想情

况。实际上完整的一元二次方程求解过程,应该考虑以下几种可能情况。

- (1) 当 $a=0$ 时,变成一元一次方程求解。
- (2) 当 $b^2-4ac=0$ 时,方程有两个相等的实根。
- (3) 当 $b^2-4ac>0$ 时,方程有两个不相等的实根。
- (4) 当 $b^2-4ac<0$ 时,方程有两个共轭复根。

例 4-11 的程序代码如下:

```

1 //求解一元二次方程
2 package main
3
4 import(
5     "fmt"
6     "math"
7 )
8
9 func main() {
10     //设 x1,x2 为两个实数根,disc 为判别式.
11     var a,b,c,disc,x1,x2,p,q float64
12     //设 cpx1,cpx2 为两个共轭复根
13     var cpx1,cpx2 complex128
14     fmt.Println("请输入一元二次方程三个系数:")
15     fmt.Scanf("%f %f %f",&a,&b,&c).
16     if a==0 {
17         fmt.Println("系数 a 为 0 不是一元二次方程!")
18     } else {
19         disc=b*b-4*a*c
20         p=-b/(2*a)
21         q=math.Sqrt(disc)/(2*a)
22         if disc==0 {
23             x1=p
24             x2=p
25             fmt.Printf("判别式等于 0 有两个相等的实根: x1=x2=%8.2f\n",p)
26         } else if disc>0 {
27             x1=p+q
28             x2=p-q
29             fmt.Printf("判别式大于 0 有两个不相等的实根: x1=%8.2f x2=%8.2f\n",x1,x2)
30         } else if disc<0 {
31             q=math.Sqrt(math.Abs(disc))/(2*a)
32             cpx1=complex(p,q)
33             cpx2=complex(p,-q)
34             fmt.Printf("判别式小于 0 有两个共轭的复根: cpx1=%8.2f cpx2=%8.2f\n",cpx1,cpx2)
35         }
36     }
37 }

```

对该例分为下面 4 种情况进行测试:

- (1) 系数 a 为 0 的情况。

从键盘输入: 0,2.0,3.0 ✓

输出结果为: 系数 a 为 0 不是一元二次方程!

(2) 判别式等于 0 的情况。

从键盘输入: 1.0,2.0,1.0 ✓

输出结果为: 判别式等于 0 有两个相等的实根: $x_1 = x_2 = -1.00$

(3) 判别式大于 0 的情况。

从键盘输入: 2.1,6.3,1.5 ✓

输出结果为: 判别式大于 0 有两个不相等的实根: $x_1 = -0.26$ $x_2 = -2.74$

(4) 判别式小于 0 的情况。

从键盘输入: 1.0,2.0,2.0 ✓

输出结果为: 判别式小于 0 有两个共轭的复根: $\text{cpx}_1 = -1.00 + 1.00i$ $\text{cpx}_2 = -1.00 - 1.00i$

4.3.2 打印中文日期信息

例 4-12 打印中文日期信息,并使用 switch 语句判断当天是星期几,并打印出来。

解决该问题要使用 time 包中的相关函数,所以要在程序开始导入 time 包。该程序的算法过程如下:

(1) 使用 Now() 函数获取系统当前时间戳。

(2) 使用时间对象的 Format() 方法,按照“年、月、日”格式输出中文日期信息。

(3) 使用时间对象的 Weekday() 方法获取星期信息,再使用 String() 方法将它转换成字符串格式。

(4) 使用 switch 语句判断并输出星期信息。

例 4-12 的程序代码如下:

```
1 //打印中文日期和星期信息
2 package main
3
4 import(
5     "fmt"
6     "time"
7 )
8
9 func main() {
10     //获取时间戳
11     t := time.Now()
12     //按年 月 日格式输出日期信息
13     fmt.Printf(t.Format("2006 年 01 月 02 日"))
14     switch t.Weekday().String() {
15     case "Sunday":
16         fmt.Printf(" 星期天")
17     case "Monday":
18         fmt.Printf(" 星期一")
19     case "Tuesday":
20         fmt.Printf(" 星期二")
```

```
21     case "Wednesday":
22         fmt.Printf(" 星期三")
23     case "Thursday":
24         fmt.Printf(" 星期四")
25     case "Friday":
26         fmt.Printf(" 星期五")
27     case "Saturday":
28         fmt.Printf(" 星期六")
29     }
30 }
```

编译并运行该程序,输出结果为:

2013 年 07 月 19 日 星期五

小结

本章主要介绍了 Go 语言的选择结构程序设计方法,包括 if 语句和 switch 语句。if 语句又分为三种形式:单分支选择结构、双分支选择结构和多分支选择结构。switch 语句也包括基本的使用方法和特殊的使用方法,比如多选项 case 语句、fallthrough 语句和无条件表达式 switch 语句。

通过这一章的学习,首先要掌握选择结构程序设计的方法,然后要掌握多种 if 语句和 switch 语句的特殊使用方法,最后就是读者如果有其他高级语言的基础,应当对比总结一下 Go 语言中的 if、switch 语句的特别之处。

习题

4.1 写一程序求 y 值(x 值由键盘输入)。

$$y = \begin{cases} \frac{\sin(x) + \cos(x)}{2} & (x \geq 0) \\ \frac{\sin(x) - \cos(x)}{2} & (x < 0) \end{cases}$$

4.2 输入一个字符,判断它如果是小写字母输出其对应大写字母;如果是大写字母输出其对应小写字母;如果是数字输出数字本身;如果是空格,输出“SPACE”;如果不是上述情况,输出“Other”。

4.3 有三个数 a、b、c,由键盘输入,输出其中最大的数。

4.4 输入一个数,判断它能否被 3 或者被 5 整除,如至少能被这两个数中的一个整除则将此数打印出来,否则不打印,编出程序。

4.5 读入 1~7 之间的某个数,输出表示一星期中相应的某一天的单词:Monday、Tuesday 等,使用 switch 语句实现。

第5章

Go 循环结构程序设计

在程序运行时,往往需要重复执行某一段程序,可以使用循环控制来实现这种设计,这种程序结构就叫循环结构(Loop Structure)。循环结构可以减少程序源代码重复书写的工作量,还可以用来描述重复执行某段算法的问题,这是程序设计中最能发挥计算机特长的程序结构。Go 语言的循环控制语句和 C 语言十分类似,但是 Go 语言没有 C 语言中的 do 或 while 循环,仅使用 for 语句就实现了所有循环控制。

5.1 for 语句

Go 语言的循环控制结构由 for 语句实现,共有三种形式:for 基本循环结构、for 条件循环结构和 for 无限循环结构。

5.1.1 for 基本循环结构

在 Go 语言中,最常见的是 for 基本循环结构(For-loop Structure),它由“for”关键字、初始化表达式、条件表达式、步进表达式和循环体组成,一般格式如下:

```
for 初始化表达式; 条件表达式; 步进表达式{
    循环体
    :
}
```

在 for 语句中,初始化表达式给循环变量赋初值,设置循环的开始位置;条件表达式用于判断是否执行本次循环;步进表达式用于在每次循环体执行后更改循环变量的值。

for 语句的执行步骤如下:

- (1) 先执行初始化表达式,给循环变量赋初值,这一步只执行一遍。
- (2) 执行条件表达式,如果结果为 true,则执行循环体中的语句,循环体中的语句执行结束,继续执行步骤(3);如果结果为 false,则结束循环,转到步骤(5)。
- (3) 执行步进表达式,修改循环变量的值。
- (4) 转回步骤(2)继续执行。
- (5) 循环结束,执行 for 循环体外的语句。

例 5-1 基本循环控制结构的使用,统计单词中的某个字符数。

```
1 //使用基本循环控制结构统计单词中的某个字符数
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var words string
10    var ch byte
11    var ln,count int
12    //从键盘格式化输入单词
13    fmt.Scanf("% s",&words)
14    fmt.Println("\n")
15    //输入要统计的字符
16    fmt.Scanf("% c",&ch)
17    //计算单词的字节长度
18    ln = len(words)
19    /*
20     算法: 从单词的第一个字符开始扫描,找到符合条件的字符,
21          计数器加 1.
22     */
23    for i := 0; i < ln; i++{
24        if byte(words[i]) == ch {
25            count++
26        }
27    }
28    fmt.Printf("There are %d %q in the %q.\n",count,ch,words)
29 }
```

编译并运行该程序,测试过程如下:

从键盘输入: internet ↵

t ↵

输出结果为: There are 2 't' in the "internet".

在例 5-1 的基本 for 循环结构中,只有一个循环变量参与循环控制。除此之外,for 语句还允许有两个或多个循环变量同时参与循环控制。在下面的例 5-2 中,就演示了一种由两个循环控制变量同时对循环进行控制的例子。

例 5-2 某学校要进行期末考试,考场内的座位共有 10 排,每排 5 个共 50 个座位。座位号范围为: 0~49,现要求按照 S 型排座位并打印出座位号。

分析该问题可知,每排座位需打印 5 张座位号,要求按 S 型排座位那么前一排座位号如果是正序,则后一排座位号必然为逆序。也即本例的核心问题是要循环打印 10 排座位号,但一排为正序,下一排就要为逆序。

所以,可使用两个循环变量对循环进行控制,比如 i 和 j。循环变量 i 控制正序打印;循

环变量j控制逆序打印。例 5-2 代码如下:

```
1 //按照 S 型排座位
2 package main
3
4 import(
5     "fmt"
6     "strconv"
7 )
8
9 func main() {
10     //定义并初始化座位为 10 行 5 列
11     var row,col int=10,5
12     //定义座位号打印字符串
13     var seatNo string
14     for k:=0; k<row; k++){
15         for i,j:=k*col,(k+1)*col-1; i<(k+1)*col && j>=k*col; i,j=i+1,j-1 {
16             if k%2==0 {
17                 seatNo+=" "+strconv.Itoa(i)
18             } else {
19                 seatNo+=" "+strconv.Itoa(j)
20             }
21         }
22         fmt.Printf("%s\n",seatNo)
23         seatNo=""
24     }
25 }
```

编译并运行该程序,输出结果为:

```
0   1   2   3   4
9   8   7   6   5
10  11  12  13  14
19  18  17  16  15
20  21  22  23  24
29  28  27  26  25
30  31  32  33  34
39  38  37  36  35
40  41  42  43  44
49  48  47  46  45
```

5.1.2 for 条件循环结构

Go 语言的另外一种循环结构是条件循环结构(Conditional-loop Structure),它类似于 C 语言中的 while 循环语句,即当型循环,它由“for”关键字、条件表达式和循环体组成,一般格式如下:

```
for 条件表达式{
    循环体
    :
}
```


例 5-3 使用条件循环控制结构近似求 π 。

基本算法：使用 $\pi/4 \approx 1 - 1/3 + 1/5 - 1/7 + \dots$ 公式求 π 的近似值，直到某一项的绝对值小于 10^{-6} 为止。

```
1 //使用条件循环控制结构近似求  $\pi$ 。
2 package main
3
4 import(
5     "fmt"
6     "math"
7 )
8
9 func main() {
10     var s int = 1
11     var n,t,pi float64 = 1.0,1.0,0
12     for math.Abs(t) > 1e-6 {
13         pi = pi + t
14         n = n + 2
15         s = -s
16         t = float64(s) / n
17     }
18     pi = pi * 4
19     fmt.Printf("pi = %10.6f\n",pi)
20 }
```

编译并运行该程序，输出结果为：

```
pi = 3.141591
```

5.1.3 for 无限循环结构

另外一种比较常见的循环结构是无限循环结构(Lnfinite-loop Structure)，和条件循环结构不同，无限循环结构退出的条件不是放在条件表达式中，而是放在循环体中，一般由 if 语句和 break 语句构成。无限循环结构直接由“for”关键字和循环体构成，一般形式如下：

```
for {
    循环体
    :
}
```

这种形式还可以写成：for true{ 循环体 } 或 for; ; { 循环体 } 的形式，后一种形式中的“;”通常会被 gofmt 命令自动清除，所以最终和 for 无限循环的基本形式一致。

由于在 for 无限循环语句中没有条件控制，在每一次循环体执行后控制条件依然为 true，那么当循环执行一定次数想要退出循环时，就要采取其他措施。比如可以在循环体中使用 if 语句进行终止循环的条件判断，当终止循环条件为 true 时，可以调用 break 语句跳出循环。

在 Go 语言中还有一种无限循环,它有初始化表达式、步进表达式,但是没有条件表达式,比如以下形式:

```
for i:=0;;i++){
    循环体
    :
}
```

例 5-4 使用 for i:=0;;i++{} 循环实现简单键盘文本输入。

```
1 //简单键盘文本输入程序
2 package main
3
4 import(
5     "bufio"
6     "os"
7     "fmt"
8 )
9
10 func main() {
11     var c byte           //存放每个按键 ASCII 码
12     var str string       //存放输入字符串流
13     //初始化标准输入设备
14     r := bufio.NewReader(os.Stdin)
15     //初始化标准输出设备
16     w := bufio.NewWriter(os.Stdout)
17     for i:=0;;i++){
18         //从标准输入设备接收字符
19         c,_ = r.ReadByte()
20         if c == 10 {
21             break        //如果是回车停止接收退出循环
22         } else {
23             //将接收到的字符写入标准输出设备
24             w.WriteByte(c)
25             w.Flush()
26             //生成字符串
27             str += string(c)
28         }
29     }
30     fmt.Printf("\n")
31     fmt.Println(str)
32 }
```

编译并运行该程序,测试过程如下:

从键盘输入: Golang is a beautiful language. ✓

输出结果为: Golang is a beautiful language.

Golang is a beautiful language.

输出结果第一行是将从标准输入设备(键盘)上接收的字符,从标准输出设备(显示终端)输出,第二行是存放在字符串中的数据,可以看出二者是一致的。

5.1.4 使用 for 语句的注意事项

使用 for 语句时,应注意以下几点:

(1) for 语句中的三个表达式之间用“;”隔开,不能使用括号括起来。

例如,下面的表达式语句是正确的:

```
for i := 0; i < 10; i++{  
}
```

下面的表达式语句是错误的:

```
for (i := 0; i < 10; i++) {  
}
```

(2) 左大括号“{”必须和 for 语句放在同一行,否则编译会出错。

例如,下面的语句“{”使用正确:

```
for i := 0; i < 10; i++{  
}
```

下面的语句“{”使用错误:

```
for i := 0; i < 10; i++  
{  
}
```

(3) 循环变量的名字尽量要短,比如 i、j、z 或 ix 等。

(4) 不要在循环体中修改循环变量的值,否则会造成意想不到的错误,这已经在很多语言中得到了证明。

(5) 如果使用多个循环变量,Go 语言不支持以“,”为间隔的多个赋值语句,必须使用平行赋值的方式来初始化多个变量。

例如,下面的多个循环变量的赋值方式是正确的:

```
for i, j := 0, 10; i < j; i, j = i+1, j-1 {  
}
```

下面的赋值方式是错误的:

```
for i := 0, j := 10; i < j; i++, j-- {  
}
```


5.1.5 for 循环嵌套结构

一个循环结构的循环体内又包含另外一个完整的循环结构,称为循环嵌套。内嵌的循环中还可以嵌套循环,就是多层循环。for 循环语句的嵌套格式如下:

```
for ; ; {
    for ; ; {
        :
    }
}
```

例 5-5 使用循环嵌套打印九九乘法表。

```
1 //使用循环嵌套打印九九乘法表
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var row,col int
10    for row=1; row<=9; row++){
11        for col=1; col<=row; col++){
12            fmt.Printf("%d*%d=%d ",row,col,row*col)
13        }
14        fmt.Printf("\n")
15    }
16 }
```

编译并运行该程序,输出结果为:

```
1*1=1
2*1=2 2*2=4
3*1=3 3*2=6 3*3=9
4*1=4 4*2=8 4*3=12 4*4=16
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

编译并运行该程序,测试过程如下:

编译并运行该程序,测试过程如下:

5.2 跳转语句

当程序需要退出当前循环,或需要转去其他分支继续执行的时候,可以使用跳转语句(Jump Statement)。Go 语言中有三个跳转语句: goto 语句、break 语句和 continue 语句。

这三条语句都可以配合标签使用。标签名是区分大小写的,若不使用会造成编译错误。`break`与`continue`配合标签使用可用于多重循环的跳出。`goto`语句是调整执行位置,与其他两个语句配合标签的结果并不相同。

5.2.1 break 语句

`break`语句是限定转向语句,它能控制程序流程跳出所在的结构,把程序流程转向所在结构之后。所以,`break`在`for`循环结构中的作用就是控制程序流程跳出循环体,即提前结束循环。`break`语句格式如下:

break

例 5-6 判断一个数是否为素数。

判断一个数是否为素数,可以采用如下算法:假如要判断数字 `num` 是否是素数,可以让 `num` 除以 $2 \sim k(k = \text{int } \sqrt{\text{num}})$ 之间的数,如果 `num` 能被 $2 \sim k$ 之间的任何一个数整除,则 `num` 不是素数;否则,`num` 是素数。

```
1 //判断素数
2 package main
3
4 import(
5     "fmt"
6     "math"
7 )
8
9 func main() {
10     var num,i,k int
11     //从键盘格式化输入要判断的数 num
12     fmt.Scanf("% d",&num)
13     //对 num 开方并取整求 k
14     k = int(math.Sqrt(float64(num)))
15     for i = 2; i <= k; i++{
16         if num % i == 0 {
17             break //num 能被 2~k 中的一个数整除,跳出循环.
18         }
19     }
20     if i > k { //num 不能被 2~k 中的任何一个数整除,则 num 是素数.
21         fmt.Printf("The number % d is a prime number.\n",num)
22     } else {
23         fmt.Printf("The number % d is not a prime number.\n",num)
24     }
25 }
```

编译并运行该程序,测试过程如下:

从键盘输入: 102 ✓

输出结果为: The number 102 is not a prime number.

从键盘输入: 13 ✓

输出结果为: The number 13 is a prime number.

在例 5-6 中, break 语句使用在一级循环结构中控制循环跳出。另外, break 语句还可以配合标签(Label), 从多级嵌套循环中跳出。例如:

```
for {
    for i:=0;i<10;i++){
        if i>3{
            break
        }
    }
}
```

该例是一个两层循环嵌套结构, 第一层是无限循环、第二层是有限循环。虽然在第二层使用了 break 语句, 但仍然是一个无限循环。因为 break 语句只能跳出第二层循环, 而无法跳出第一层循环。

要使 break 语句能控制程序流程跳出上述的循环嵌套结构, 可以采用在 break 语句后加一个跳转标签的方式。格式如下:

```
LABEL:
:
break LABEL
```

使用 break 加跳转标签的方式将上述程序代码改写为:

```
LABEL1:
    for {
        for i:=0;i<10;i++){
            if i>3{
                break LABEL1
            }
        }
    }
```

这时, 如果触发 break 语句, 程序流程将跳出和标签同级的这一层循环。比如该例中, 第一层无限循环和标签“LABEL1”同级, 那么 break 语句将控制程序流程跳出第一层的无限循环。

5.2.2 continue 语句

continue 语句用于 for 循环结构中, 它的作用是结束本次循环, 即跳过循环体中下面尚未执行的语句, 接着进行下一次是否执行循环体的判定。其格式如下:

```
continue
```

break 语句和 continue 语句的区别在于: continue 语句只结束本次循环, 而不是终止整个循环; 而 break 语句则是终止整个循环过程, 不再判断执行循环的条件是否成立。

例 5-7 把 10~30 之间能被 3 整除的数输出。

```
1 //continue 语句的使用
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var n int
10    for n = 10; n < 30; n++{
11        if n%3 != 0 {
12            continue //不能被 3 整除,终止本次循环,返回条件测试.
13        }
14        fmt.Printf("%d ",n)
15    }
16    fmt.Printf("\n")
17 }
```

编译并运行该程序,输出结果为:

```
12 15 18 21 24 27
```

当 n 不能被 3 整除时,执行 `continue` 语句,结束本次循环,即跳过 `Printf` 函数语句。只有 n 能被 3 整除时,才执行 `Printf` 函数语句。

和 `break` 语句一样,`continue` 语句也可以配合标签(Label),终止多级嵌套循环中的某次循环。格式如下:

```
LABEL:
:
continue LABEL
```

例如:

```
for i := 0; i < 10; i++{
    for {
        continue
    }
}
fmt.Println("OK")
```

该例为两层循环嵌套结构,第一层是有限循环,第二层是无限循环。但是上述代码永远不会执行到 `Println` 语句,因为 `continue` 语句只能作用于第二层循环,所以程序陷入到无限循环中。

`continue` 配合标签将上述代码修改如下:

```

LABEL1:
    for i := 0; i < 10; i++{
        for {
            continue LABEL1
        }
    }
    fmt.Println("OK")
}

```

这时,每一次 continue 语句程序将跳到标签“LABEL1”处继续执行,显然第一层循环会被执行结束,Println 语句也会被执行。

5.2.3 goto 语句

和 break、continue 语句不同,goto 语句天生就要和标签(Label)配合来使用。goto 语句可以调整程序执行的位置,它可以让程序无条件跳转到一个标签(Label)之处继续执行。其格式如下:

```

LABEL:
:
goto LABEL

```

goto 语句和标签配合使用时,标签的后面要加“:”,另外标签是区分大小写的,若不使用会造成编译错误,使用时要注意。

在 5.1.2 节中讲过,for 条件循环语句类似于 C 语言中的 while 循环语句,即当型循环。那么 goto 语句与标签和 if 语句配合,则可以实现类似于 C 语言中的 do while,即直到型循环。

例 5-8 goto 语句实现直到型循环,计算 $\sum(1\sim 100)$ 的累加和。

```

1 //goto 语句实现直到型循环计算 1~100 的累加和
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var i, sum = 1, 0
10 HERE:
11     sum = sum + i
12     i++
13     if i <= 100 {
14         goto HERE
15     }
16     fmt.Println(sum)
17 }

```

编译并运行该程序,输出结果为:

5050

总结前面的知识可以发现, `break`、`continue`、`goto` 语句都可以和标签配合使用,但作用有所区别。通常情况下,如果不使用标签 `break`、`continue` 语句只能跳出本层循环。配合标签使用以后, `break`、`continue` 语句就可以跳出多重循环了,这时就不必使用 `goto` 语句了。分析如下代码:

```
for {
    for i := 0; i < 5; i++ {
        if i > 3 {
            break
        }
    }
}
```

上述代码很显然永远不会执行完,因为 `break` 只能跳出第二层循环,而第一层循环仍然是一个无限循环。再看下面的代码:

```
LABEL1:
    for {
        for i := 0; i < 5; i++ {
            if i > 3 {
                break LABEL1
            }
        }
    }
}
```

在 `break` 后加上标签“`LABEL1`”后,程序就会跳出和“`LABEL1`”同级的这一层循环,也就是第一层循环,这时就可以成功地跳出多层循环。

如果将上述代码中的 `break` 改成 `goto`,程序还能跳出无限循环吗? 答案是否定的。因为 `goto` 只能改变程序执行的位置,当程序 `goto` 到“`LABEL1`”后,会接着从第一层 `for` 循环继续执行,仍然是无限循环。

有一种处理方法,可以使 `goto` 语句也能跳出上述无限循环,做法是将标签“`LABEL1`”放到第一层 `for` 循环后面,而不是之前。看下面的代码:

```
for {
    for i := 0; i < 5; i++ {
        if i > 3 {
            goto LABEL1
        }
    }
}
LABEL1:
```


最后对于上述例子,如果使用 `continue` 语句和标签配合使用,那么它的第一层循环必须是有限循环,这样才能保证程序能够终止。

5.3 for range 语句

在 Go 语言中, `for` 语句还可以和 `range` 关键字配合,组成 `for range` 语句, `for range` 语句可以对数组(Array)、切片(Slice)等对象的元素进行遍历。在 Go 语言中, `range` 可以看作是一个迭代器,当它被调用时,它会从所遍历的数组返回一个键值对(Key-Value Pair)。

基于不同的数据对象, `range` 返回不同的键值对。当对数组或切片进行遍历时, `range` 返回数组下标作为键(Key), 数组下标所对应的元素作为值(Value), 关于数组和切片的知识将在第 6 章进行讲解。

由于字符串也可以看作是字符数组,所以这里以字符串为例,来学习 `for range` 语句的用法。

例 5-9 `for range` 语句对数据对象进行遍历。

```
1 //for range 语句对数据对象进行遍历
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var str string = "Golang"
10    //设 k 为键, v 为值
11    for k, v := range str {
12        fmt.Printf("s[ %d] = %c\n", k, v)
13    }
14 }
```

编译并运行该程序,输出结果为:

```
s[0] = G
s[1] = o
s[2] = l
s[3] = a
s[4] = n
s[5] = g
```

除了前面所介绍的, `for range` 语句还可以对字典(Map)、通道(Channel)进行迭代操作。有时用户可能只需要 `range` 返回的键值对中的“键”或者“值”,那么可以使用空白操作符“_”作为占位符,过滤掉不需要的数据。

例如,如果只需要“值”不需要“键”,例 5-9 中 `for range` 语句可以修改如下:

```
for _, v := range str {  
    :  
}
```

如果没有“_”作为占位符,那么 range 会把第一个产生的数赋给 v。而有了占位符,range 首先将 key 赋给“_”,然后再将其丢掉。最后 range 会将 value 赋给 v,这样才能获得用户所需要的数据。

如果只需要“键”不需要“值”,例 5-9 中 for range 语句可以修改如下:

```
for k, _ := range str {  
    :  
}
```

解决上面的问题,也可以直接将“值”丢掉。这是因为 k 本身就在第一位,会首先获取数据。无论有没有占位符,k 都将获取想要的数。代码如下:

```
for k := range str {  
    :  
}
```

5.4 循环控制程序举例

前面介绍了 for 语句和跳转语句的基本用法,本节将列举几个典型例子,进一步探讨循环控制结构在解决现实问题中的作用。

5.4.1 多重循环嵌套应用举例

在现实应用中,单重循环虽然逻辑简单,但难于解决一些复杂问题。要解决一些复杂问题,往往需要使用多重循环。在下面的例 5-10 中,将使用多重循环嵌套解决一个数字组合问题。

例 5-10 使用 1、2、3、4 这 4 个数字,组合成互不相同其无重复的三位数,并按照一行 5 个输出符合条件的组合数。

算法分析: 1、2、3、4 这 4 个数字分别可以填在百位、十位、个位,按照排列的方法先确定百位、再确定十位、最后确定个位,然后将不满足条件的排列去掉即可。所以可以使用三重循环嵌套实现上述算法。

例 5-10 代码如下:

```
1 //数位重组  
2 package main  
3  
4 import(  
5     "fmt"  
6     "strconv"
```

```

7 )
8
9 func main() {
10     //整型变量 num 用来存储生成的三位数
11     var num, col int
12     //字符串 numStr 用来存数位
13     var numStr string
14     //三重循环嵌套实现该算法
15     for i := 1; i < 5; i++{
16         for j := 1; j < 5; j++{
17             for k := 1; k < 5; k++{
18                 if i != k && i != j && j != k { //确保个位、十位、百位互不相同
19                     numStr = strconv.Itoa(i) + strconv.Itoa(j) + strconv.Itoa(k)
20                     num, _ = strconv.Atoi(numStr)
21                     fmt.Printf(" %d", num)
22                     col++
23                     //控制每行输出 5 个数
24                     if col == 5 {
25                         fmt.Printf("\n")
26                         col = 0
27                     }
28                 }
29             }
30         }
31     }
32 }

```

编译并运行该程序,输出结果为:

```

123 124 132 134 142
143 213 214 231 234
241 243 312 314 321
324 341 342 412 413
421 423 431 432

```

5.4.2 无限循环和跳转语句应用举例

现实中许多问题的解决都需要使用无限循环,比如例 5-4 使用键盘输入的例子。另外,无限循环还要和跳转语句相互配合,这样才能有效控制程序能够执行结束。例 5-11 就是这样一个典型例子,它利用无限循环和 goto 语句的巧妙配合,实现了一个猜数字游戏。

例 5-11 设定谜底数字为 36,设计一个猜数字游戏,在游戏过程中提示玩家数字过大还是过小,最后统计玩家共猜了多少次才猜中。

算法分析: 该程序需要两个主要流程“输入流程”和“猜字流程”,而且这两个流程都要使用无限循环结构。该程序设计的关键是有效控制这两个无限循环,程序控制过程如下。

(1) 初始化 I/O 设备: 初始化输入、输出设备,初始化工作结束程序进入(2)继续执行。

(2) 输入流程: 如果接收的是正常字符,循环继续;如果接收到的是“回车”或“换行”,使用 break 跳出循环,程序进入(3)继续执行。

(3) 猜字流程: 如果答案正确, 使用 break 跳出循环, 程序进入(4)继续执行; 如果答案不正确, 提示是“过小”还是“过大”, 使用 goto 语句跳转到(2)继续执行。

(4) 输出统计结果, 程序退出。

例 5-11 代码如下:

```
1 //猜数字游戏
2 package main
3
4 import(
5     "bufio"
6     "fmt"
7     "os"
8     "strconv"
9 )
10
11 const GOAL int = 36                                //设置谜底
12
13 func main() {
14     var data int                                    //存储输入数字
15     var count int                                    //统计用了多少次才猜中
16     var c byte                                       //存放每个按键 ASCII 码
17     var str string                                   //存放输入字符串流
18     fmt.Println("请输入一个正整数: ")
19 LABEL1:
20     r := bufio.NewReader(os.Stdin)                  //初始化标准输入设备
21     w := bufio.NewWriter(os.Stdout)                 //初始化标准输出设备
22     /*
23         键盘输入处理流程
24     */
25     for i := 0; ; i++{
26         c, _ = r.ReadByte()
27         if c == 10 || c == 13 {                      //如果是回车或换行则退出
28             break
29         } else {
30             w.Flush()
31             str += string(c)
32         }
33     }
34     /*
35         猜数字流程
36     */
37     for {
38         data, _ = strconv.Atoi(str)
39         if data > GOAL {
40             fmt.Printf("数字 %d 有点大, 请再试一试...\n", data)
41             count++
42             str = ""
43             goto LABEL1
44         } else if data < GOAL {
```

```
45         fmt.Printf("数字 %d 有点小,请再试一试...\n",data)
46         count++
47         str = ""
48         goto LABEL1
49     } else {
50         fmt.Println("恭喜你,猜中了!")
51         break
52     }
53 }
54 fmt.Printf("你一共猜了 %d 次才猜对了.",count)
55 }
```

编译并运行该程序,测试过程如下:

请输入一个正整数:

10 ✓

数字 10 有点小,请再试一试...

50 ✓

数字 50 有点大,请再试一试...

30 ✓

数字 30 有点小,请再试一试...

40 ✓

数字 50 有点大,请再试一试...

36 ✓

恭喜你,猜中了!

你一共猜了 5 次才猜对了.

5.4.3 for range 语句应用举例

通过 5.3 节的学习了解到,在 Go 语言中,使用 for range 语句遍历诸如数组、字符串类型的数据对象非常有效。在下面的例 5-12 中,将使用 for range 遍历用户输入的字符串,并统计字符串中字符、数字的个数。

例 5-12 通过键盘输入一组字符串,统计其中英文字符、阿拉伯数字和其他字符的个数,最后输出统计结果。

算法分析: 可以使用 for range 语句对用户输入的字符串进行遍历,然后对每一个读取到的“Value”值进行判断,如果 Value 值属于区间 $48 \leq \text{Value} \leq 57$,则为数字;如果属于区间 $65 \leq \text{Value} \leq 90$ 或 $97 \leq \text{Value} \leq 122$,则为英文字符;否则,就为其他字符。

例 5-12 代码如下:

```
1 //统计字符串中的字符、数字个数
2 package main
3
4 import(
5     "fmt"
```



```
6 )
7
8 func main() {
9     //str 用于存储输入字符串
10    var str string
11    //下列变量分别用于英文字符,数字,其他字符个数统计
12    var charCount,numCount,otherCount int
13    fmt.Println("请输入字符串...")
14    fmt.Scanf("%s",&str)
15    for _,v := range str {
16        if v >= 48 && v <= 57 {
17            numCount++
18        } else if (v >= 65 && v <= 90) || (v >= 97 && v <= 122) {
19            charCount++
20        } else {
21            otherCount++
22        }
23    }
24    fmt.Printf("共有 %d 个数字\n",numCount)
25    fmt.Printf("共有 %d 个英文字母\n",charCount)
26    fmt.Printf("共有 %d 个其他字符\n",otherCount)
27 }
```

编译并运行该程序,测试过程如下:

请输入字符串...

12afgf24,.0--afjklgj35t,>af ✓

共有 7 个数字

14 个英文字母

6 个其他字符

6.1.1 数组的声明

小结

在声明数组时,必须指定数组名、数组长度和数组类型,格式如下:

本章主要介绍了 Go 语言的循环控制结构程序设计方法,包括基本循环结构、条件循环结构和无限循环结构。另外,还介绍了三种跳转语句: break 语句、continue 语句和 goto 语句。最后介绍了一种能对数组、切片、字典进行遍历的特殊语句: for range 迭代器。

通过这一章的学习,了解到 Go 程序中的各种循环控制都是由 for 语句完成的。所以,读者必须掌握各种 for 循环控制结构程序设计的方法。最后应该明确一点,一个灵活的循环控制程序是由 for 语句和跳转语句共同完成的。Go 语言给 break、continue 语句赋予了新的特征,比如和标签配合使用。另外,在 Go 语言中 goto 语句获得了新生,可以被广泛使用。在其他高级语言中,这是不可想象的。

例如:

习题

- 5.1 求 1~100 之间的奇数之和、偶数之积。
- 5.2 定义一个整型数组,元素个数为 10,打印出数组元素中的最大数、最小数。
- 5.3 打印出所有的“水仙花数”,所谓“水仙花数”是指一个三位数,其各位数字立方和等于该数本身。例如,153 是一个水仙花数,因为 $153=1^3+5^3+3^3$ (要求分别用一重循环和三重循环实现)。
- 5.4 FizzBuzz 是英国学校里常玩的游戏,从 1 数到 100,如果遇见了 3 的倍数要说“Fizz”,如果遇见了 5 的倍数要说“Buzz”,如果既是 3 的倍数又是 5 的倍数要说“FizzBuzz”。编写一个程序,按游戏要求打印数字 1~100。
- 5.5 若一个数恰好等于它的平方数的右端,则这个数称为同构数。如 5 的平方是 25,5 是 25 中的右端的数,5 就是同构数。找出 1~1000 之间的全部同构数。
- 5.6 一球从 100m 高度自由落下,每次落地后反跳回原来高度的一半,再落下,求它在第 10 次落地时,共经过多少米? 第 10 次反弹多高?
- 5.7 某铁路线上共有 10 个车站,如果每两个车站之间需要一种车票,问共需准备多少种车票?
- 5.8 从键盘输入字符串 str,使用 for range 语句遍历该字符串中的每一个字符,并打印输出。

第6章

数组、切片和字典

前面几章的内容,本教材使用的都是 Go 语言的基本数据类型(值类型),比如:布尔型、整型、浮点型、字符型等。除此之外,Go 语言还提供了构造类型和引用类型。

值类型(Value-type),是一种由类型的实际值表示的数据类型。如果向一个变量分配值类型,则该变量将被赋以全新的值副本。这与引用类型不同,在引用类型中,赋值时不创建副本。Go 语言的值类型包括布尔型、整型、浮点型、字符型、复数型等。

构造类型(Structure-type),是由基本数据类型按照一定的规则组成,又称为“导出类型”。Go 语言的构造类型包括数组(Array)、结构体(Struct)和字符串(String)。

引用类型(Reference-type),是由类型的实际值引用(类似于指针)表示的数据类型。如果为某个变量分配一个引用类型,则该变量将引用(或“指向”)原始值,不创建任何副本。Go 语言引用类型包括切片(Slice)、字典(Map)和通道(Channel)。

6.1 数组

数组是一组具有相同类型和名称的变量的集合,数组中的每一个值称为数组元素(Array-element),每个数组元素都有一个编号,这个编号叫做数组下标(Array-index),可以通过下标来区别这些元素。数组元素的个数有时也称为数组的长度。

6.1.1 数组的声明

在声明数组时,必须指定数组名、数组长度和数组类型,格式如下:

```
var arrayName [arraySize] dataType
```

说明:

- (1) 数组名的命名规则和变量名相同,遵循标识符命名规则。
- (2) 在定义数组时,需要指定数组中元素的个数,即数组长度。例如, `a [5]int` 表示整型数组 `a` 有 5 个元素。对数组元素的访问通过数组下标来进行,数组下标通常由“0”开始,数组 `a` 的 5 个元素是: `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`。注意,不存在数组元素 `a[5]`。
- (3) 一般情况下,数组所有元素的类型必须相同,可以是 Go 语言的任何基本数据类型。例如:

```
var arr1 [5] int
```

在该例中,声明了一个长度为 5 的整型数组 arr1,数组 arr1 中的元素在内存中的存储模型如图 6-1 所示。

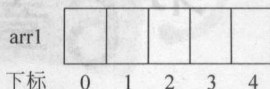


图 6-1 数组 arr1 的存储模型

例 6-1 数组的定义。

```
1 //数组的定义
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var i int
10    var arr1 [5]int
11    for i=0; i<=4; i++){
12        arr1[i]=i+1
13    }
14    fmt.Println(arr1)
15 }
```

编译并运行该程序,输出结果为:

```
[1 2 3 4 5]
```

6.1.2 数组的初始化

当数组定义好以后,如果没有给数组元素指定值,则所有元素被自动初始化为 0。对数组元素的初始化,可以使用以下几种方式实现。

(1) 在定义数组时对数组元素赋初值。例如:

```
var a=[10]int{1,2,3,4,5,6,7,8,9,10}
```

在给数组元素赋初值时,要将初值依次放在一对“{}”内,上例的定义和初始化后,a[0]=1,a[1]=2,a[2]=3,a[3]=4,a[4]=5,a[5]=6,a[6]=7,a[7]=8,a[8]=9,a[9]=10。

(2) 可以只给一部分元素赋初值。例如:

```
var b=[10] int{1,2,3,4,5}
```

该例中数组一共有 10 个元素,只给前 5 个元素赋了初值,则后面 5 个元素默认初始化为“0”。

(3) 可以由初始化列表决定数组长度。例如:

```
var c=[...] int{1,2,3,4,5}
```


该例中数组长度为“...”标识符,即没有指定数组长度,数组长度由初始化列表决定,在“{ }”中给定了5个元素的初值,所以数组长度为5。注意,使用这种方式时,不能省略标识符“...”,否则就变成切片了。

(4) 还可以按下标初始化元素。例如:

```
var d = [10] int{2:4,5:7}
```

该例中 $d[2]=4$, $d[5]=7$,其他数组元素被初始化为0。

例 6-2 数组的初始化。

```
1 //数组的初始化
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var a [10]int
10    var b = [10]int{1,2,3,4,5,6,7,8,9,10}
11    var c = [10]int{1,2,3,4,5}
12    var d = [10]int{2: 4,5: 7}
13    fmt.Println(a)
14    fmt.Println(b)
15    fmt.Println(c)
16    fmt.Println(d)
17 }
```

编译并运行该程序,输出结果为:

```
[0 0 0 0 0 0 0 0 0 0]
[1 2 3 4 5 6 7 8 9 10]
[1 2 3 4 5 0 0 0 0 0]
[0 0 4 0 0 7 0 0 0 0]
```

6.1.3 数组元素的访问和遍历

数组必须先定义,然后访问。数组元素只能逐个访问,而不能一次访问所有元素。数组元素的访问形式为:

```
arrayName [arrayIndex]
```

数组元素的下标从“0”开始,可以是整型常量或整型表达式。例如 $a[6]$ 和 $a[2*3]$ 表示的是同一个数组元素。

一个包含4个元素的二维数组,使用多维数组时要注意,多维数组的每一行长度必须一致。

例 6-3 打印 Fibonacci 数列的前 20 项,要求一行输出 5 项。

```
1 //打印 Fibonacci 数列的前 20 项
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var f = [20]int{1,1}
10    for i := 2; i < 20; i++){
11        f[i] = f[i-2] + f[i-1]
12    }
13    for i := 0; i < 20; i++){
14        if i%5 == 0 {
15            fmt.Printf("\n")
16        }
17        fmt.Printf("f[ %2d] = %4d ", i, f[i])
18    }
19 }
```

编译并运行该程序,输出结果为:

```
f[ 0] =   1  f[ 1] =   1  f[ 2] =   2  f[ 3] =   3  f[ 4] =   5
f[ 5] =   8  f[ 6] =  13  f[ 7] =  21  f[ 8] =  34  f[ 9] =  55
f[10] =  89  f[11] = 144  f[12] = 233  f[13] = 377  f[14] = 610
f[15] = 987  f[16] = 1597 f[17] = 2584 f[18] = 4181 f[19] = 6765
```

另外,在第 5 章了解到,在 Go 语言中还可以使用 range 关键字,以方便地遍历数组中的元素。所以,上例还可以改写成:

```
func main() {
    var f = [20]int{1,1}
    for i := 2; i < 20; i++){
        f[i] = f[i-2] + f[i-1]
    }
    for i, v := range f {
        if i%5 == 0 {
            fmt.Printf("\n")
        }
        fmt.Printf("f[ %2d] = %4d ", i, v)
    }
}
```

从上面的代码可以看到,range 具有两个返回值,第一个返回值 i 是数组的下标,第二个返回值 v 是元素的值。

6.1.4 多维数组

多维数组(Multidimensional-array),是指它的每一个元素也是类型相同的一维数组时,便构成了多维数组。所谓数组的类型相同是指数组大小、元素类型相同。数组的维数是指数组的下标个数,一维数组只有一个下标,二维数组有两个下标,多维数组则有多个下标。例如:

```
var a[3][4]int
```

这条语句,数组 a 被定义成了二维的。也可以把数组 a 看作是一个特殊的一维数组,它的每一个元素又是一个一维数组。比如本例数组 a 有三个元素: a[0]、a[1]、a[2],它的每个元素又是一个包含 4 个元素的一维数组,如下:

```
a[0]: a[0][0] a[0][1] a[0][2]  
a[1]: a[1][0] a[1][1] a[1][2]  
a[2]: a[2][0] a[2][1] a[2][2]
```

二维数组 a 可以看作是一个 3 行×4 列的矩阵,它在内存中是按行存放的,即在内存中先顺序存放第一行的元素,再存放第二行的元素,如例 6-4。

例 6-4 二维数组。

```
1 //二维数组  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7  
8 func main() {  
9     var a = [3][4]int{{1,2,3},{4,5},{6}}  
10    fmt.Println(a)  
11 }
```

编译并运行该程序,输出结果为:

```
[[1 2 3 0] [4 5 0 0] [6 0 0 0]]
```

该例中,数组的第一行初始化了三个元素,第 4 个元素默认为 0;第二行初始化了两个元素,其他两个元素默认为 0;第三行初始化了一个元素,剩下的三个元素默认为 0。

有了二维数组的例子,定义多维数组就非常容易了,可以按照同一种规则进行。例如:

```
var f[3][4][5]float64
```

这条语句定义数组 f 是三维数组,它有三个元素: f[0]、f[1]、f[2],它的每个元素又是一个包含 4 个元素的二维数组。使用多维数组时要注意,多维数组的每一行长度必须一致,

这和切片不一样(见 6.2 节)。

例 6-5 有一个 3×4 的整型矩阵,找出值最大的元素,并输出行号和列号。

```
1 //找出二维数组最大元素并输出行号和列号
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var i, j, row, col, max int
10    var a = [3][4]int{{1, 7, 3, 12}, {4, 9, 5, 19}, {8, 22, 6, 10}}
11    max = a[0][0]
12    for i = 0; i <= 2; i++{
13        for j = 0; j <= 3; j++{
14            if a[i][j] > max {
15                max = a[i][j]
16                row = i
17                col = j
18            }
19        }
20    }
21    fmt.Printf("max = %d, row = %d, col = %d\n", max, row, col)
22 }
```

编译并运行该程序,输出结果为:

```
max = 22, row = 2, col = 1
```

需要特别注意的是,在 Go 语言中数组是一个值类型(Value Type),所有值类型变量在赋值或作为参数传递时都将产生一次复制动作。如果将数组作为函数的参数类型,则在函数调用时该参数将发生数据复制。因此,在函数体中无法修改数组的内容,因为函数内操作的只是数组的一个副本。

6.2 切片

在 Go 语言中,切片(Slice)是数组的一个引用,它会生成一个指向数组的指针,并通过切片长度关联到底层数组部分或者全部元素。切片还提供了一系列对数组的管理功能(append、copy),可以随时动态扩充存储空间,并且可以被随意传递而不会导致所管理的数组元素被重复复制。根据以上特征,切片通常被用来实现变长数组,而且操作灵活。在 Go 语言中,切片的数据结构原型定义如下:

```
src/pkg/runtime/runtime.h
```

```
struct Slice
```

```
{
    //must not move anything
    byte * array; //actual data
    unit32 len; //number of elements
    unit32 cap; //allocated number of elements
};
```

由切片数据结构的原型定义可以看到,它抽象为以下三个部分:

(1) 指向被引用的底层数组的指针。

(2) 切片中元素的个数。

(3) 切片分配的存储空间。

6.2.1 切片的声明与创建

切片声明与创建的方法有三种:基于底层数组创建、直接创建或使用 `make()` 函数创建。

1. 基于底层数组创建切片

在创建切片时,可以基于一个底层数组,切片可以只使用数组的一部分元素或所有元素,甚至可以创建一个比底层数组还要大的数组切片,因为切片可以动态增长。创建切片的格式如下:

```
var sliceName []dataType
```

说明:

(1) 切片名的命名规则和变量名相同,遵循标识符命名规则。

(2) 在创建切片时,不要指定切片的长度。

(3) 切片的类型可以是 Go 语言的任何基本数据类型。

例如:

```
var slice1 [] int
```

上例中定义了一个整型切片 `slice1`,注意不要指定切片长度,如果指定了长度就成了定义数组了。当一个切片定义好以后,如果还没有被初始化,默认值为 `nil`,而且切片的长度为 0。切片的长度可以使用内置函数 `len()` 获取,还可以使用内置函数 `cap()` 获取切片的内存容量。

所以,当一个切片定义好以后,还要对切片进行初始化,这样切片才可用。对于上例,假如已经定义了一个整型数组 `array1`,则切片 `slice1` 的初始化方式如下:

```
slice1 = array1[start : end]
```

从上式可以看到,Go 语言支持以 `array1[start: end]` 的方式基于一个底层数组来生成切片,即切片引用的数组元素由 `array1[start]` 到 `array1[end]`,但不包含 `array1[end]`。比

如: `array1[:5]`表示引用数组的前5个元素, `array1[5:]`表示引用从 `array1[5]`开始的所有元素, `array1[4:7]`表示引用 `array1[4]`、`array1[5]`、`array1[6]`这三个元素。

如果要引用底层数组的所有元素,可采用的方式如下:

```
slicel = array1
slicel = array1[:]
slicel = array1[0:len(array1)]
```

第一种方式直接采用“=”操作符,第二种方式其实就是指引用范围是从第一个元素到最后一个元素,第三种方法和第二种方法实质一样,只不过是使用 `len()` 函数计算出数组的长度,作为“end”。

例 6-6 基于底层数组创建切片。

```
1 //基于底层数组创建切片
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     //先定义并初始化底层数组
10    var array1 = [10]int{1,2,3,4,5,6,7,8,9,10}
11    //注意切片类型和底层数组类型保持一致
12    var slicel []int
13    //部分引用的三种形式
14    slicel = array1[:5]
15    slice2 := array1[5:]
16    slice3 := array1[4:7]
17    //全部引用的三种形式
18    slice4 := array1
19    slice5 := array1[:]
20    slice6 := array1[0:len(array1)]
21    fmt.Println(slicel)
22    fmt.Println(slice2)
23    fmt.Println(slice3)
24    fmt.Println(slice4)
25    fmt.Println(slice5)
26    fmt.Println(slice6)
27 }
```

编译并运行该程序,输出结果为:

```
[1 2 3 4 5]
[6 7 8 9 10]
[5 6 7]
[1 2 3 4 5 6 7 8 9 10]
[1 2 3 4 5 6 7 8 9 10]
[1 2 3 4 5 6 7 8 9 10]
```


2. 直接创建切片

直接创建切片,即在定义切片的同时初始化切片元素,例如:

```
var slice1 = []int{1,2,3,4,5}
```

上式定义了一个整型切片 slice1,同时给它初始化了 5 个元素。

3. 使用 make 函数创建切片

在 Go 语言中,并不是必须基于底层数组才能创建切片,内置函数 make() 可以用于灵活地创建切片。例如:

```
var slice1 = make([]int,5)
```

上式创建了一个有 5 个元素的整型切片 slice1,元素的初值为 0。

在使用 make() 函数创建切片时,还可以为切片元素预留给存储空间。例如:

```
var slice1 = make([]int,5,10)
```

上式表示,创建整型切片 slice1,元素个数为 5,元素初值为 0,并预留 10 个元素的存储空间。

例 6-7 直接和使用 make 创建切片。

```
1 //直接和使用 make 创建切片
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     //直接创建
10    var slice1 = []int{1,2,3,4,5}
11    //使用 make 创建
12    var slice2 = make([]int,5)
13    //使用 make 创建并预留元素存储空间
14    var slice3 = make([]int,5,10)
15    fmt.Printf("len= %2d cap= %2d %v\n",len(slice1),cap(slice1),slice1)
16    fmt.Printf("len= %2d cap= %2d %v\n",len(slice2),cap(slice2),slice2)
17    fmt.Printf("len= %2d cap= %2d %v\n",len(slice3),cap(slice3),slice3)
18 }
```

编译并运行该程序,输出结果为:

```
len= 5 cap= 5 [1 2 3 4 5]
len= 5 cap= 5 [0 0 0 0 0]
len= 5 cap= 10 [0 0 0 0 0]
```

通过输出结果可以看出,切片 slice1 的长度为 5,存储容量为 5;切片 slice2 的长度为 5,存储容量为 5;切片 slice3 的长度为 5,存储容量为 10。

6.2.2 切片元素的访问和遍历

切片元素的遍历和数组元素的遍历一样,要通过元素下标访问,另外也可以使用关键字 range 遍历所有切片元素。切片元素访问的一般格式如下:

```
sliceName [sliceIndex]
```

例 6-8 切片元素的遍历。

```
1 //切片元素的遍历
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var slice1 = []int{1,2,3,4,5}
10    //使用下标访问切片元素
11    for i := 0; i <= 4; i++{
12        fmt.Printf("slice1[ %d] = %d ",i,slice1[i])
13    }
14    fmt.Printf("\n")
15    //使用 range 遍历所有元素
16    for i,v := range slice1 {
17        fmt.Printf("slice1[ %d] = %d ",i,v)
18    }
19 }
```

编译并运行该程序,输出结果为:

```
slice1[0]=1 slice1[1]=2 slice1[2]=3 slice1[3]=4 slice1[4]=5
slice1[0]=1 slice1[1]=2 slice1[2]=3 slice1[3]=4 slice1[4]=5
```

通过例 6-8 代码可以看出,range 表达式有两个返回值,第一个是元素的下标,第二个是元素的值。使用 range 对切片进行遍历,代码显得更简洁易懂。

6.2.3 切片的操作

在 Go 语言中,切片中的元素是可以动态增加、删除的,所以切片操作起来比数组更加强大、灵活。

1. 切片元素的增加

可以使用 append()函数向切片尾部添加新元素,这些元素保存到底层数组。append 并不会影响原来切片的属性,它返回变更后新的切片对象。

与数组相比,除了都有长度(length)以外,切片多了一个容量(capacity)的概念,即切片中元素的个数和分配的存储空间是两个不同的值,如例 6-7 的输出结果。如果在增加新元素时超出 cap 的限制,则底层会重新分配一块“够大”的内存,一般来说是将原来的内存空间扩大二倍,然后再将数据从原来的内存复制到新的内存块。

例 6-9 给切片增加元素。

```
1 //使用 append 函数增加切片元素
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     //使用 make 创建切片 len = 3, cap = 6
10    var slice1 = make([]int, 3, 6)
11    //使用 append 给切片增加元素且未超出 cap
12    slice2 := append(slice1, 1, 2, 3)
13    //使用 append 给切片增加元素并且超出 cap
14    slice3 := append(slice1, 1, 2, 3, 4)
15    fmt.Printf("len = %d cap = %d %v\n", len(slice1), cap(slice1), slice1)
16    fmt.Printf("len = %d cap = %d %v\n", len(slice2), cap(slice2), slice2)
17    fmt.Printf("len = %d cap = %d %v\n", len(slice3), cap(slice3), slice3)
18 }
```

编译并运行该程序,输出结果为:

```
len = 3  cap = 6  [0 0 0]
len = 6  cap = 6  [0 0 0 1 2 3]
len = 7  cap = 12 [0 0 0 1 2 3 4]
```

从输出结果可以看出,append 只是往切片尾部增加元素,如果不超出容量,增加新元素不会改变原来切片的属性,slice1 和 slice2 的容量都是 6。如果超出容量,则分配一个是原来容量 2 倍的新的内存块,再复制数据到新的内存块,所以 slice3 的容量为 slice1 的二倍是 12。

2. 切片元素的复制

使用切片长时间引用“超大”的底层数组,会导致严重的内存浪费。可以新建一个小的 slice 对象,然后将所需的数据复制过去。函数 copy() 可以在切片之间复制元素,能够复制的数量取决于复制方和被复制方的长度值,通常取最小值。需要注意的是,在同一底层数组的不同切片间复制元素时,元素位置会发生重叠。

例 6-10 切片元素的复制。

```
1 //切片的复制
2 package main
```



```
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var slice1 = []int{1,2,3,4,5,6,7,8,9,10}
10    var slice2 = make([]int,3,5)
11    var n int
12    //只能复制三个元素
13    n = copy(slice2,slice1)
14    fmt.Println(n,slice2,len(slice2),cap(slice2))
15    //slice3 和 slice1 指向同一底层数组
16    slice3 := slice1[3:6]
17    //复制后元素重叠
18    n = copy(slice3,slice1[1:5])
19    fmt.Println(n,slice1,slice3)
20 }
```

编译并运行该程序,输出结果为:

```
3 [1 2 3] 3 5
3 [1 2 3 2 3 4 7 8 9 10] [2 3 4]
```

通过输出结果可以看出,在将 slice1 复制到 slice2 时,由于 slice2 的长度最小为 3,所以只能将 slice1 的前三个元素复制给 slice2。而将 slice1 复制到 slice3 时,由于 slice1 和 slice3 指向同一个底层数组,所以复制后元素重叠。slice3 刚创建时,它引用的是底层数组的 [4,5,6]三个元素,复制后 slice1 将 [2,3,4]三个元素复制给 slice3,所以最后 slice3 的元素 [2,3,4]覆盖了 slice1 的元素 [4,5,6]。

6.3 字典

在 Go 语言中,Map 是一种特殊的数据结构,它由一对无序的数据项组成,被称为键值对(Key-value Pair)。其中的一项是键(Key),另外一项是值(Value),Map 通过把键映射到值来进行访问,这种方式可以加快数据查找的速度。所以,Map 通常也被称作字典(Dictionary)或哈希表(Hash table),本教材统一称为“字典”。

6.3.1 字典的声明

字典也是一种引用数据类型,在声明字典时,除了要定义字典名,还要指定“键”类型和“值”类型,“键”类型要使用一对方括号“[]”括起来。字典一般声明格式如下:

```
var mapName map[keyType]valueType
```

说明:

- (1) 字典名的命名规则和变量名相同,遵循标识符命名规则。
- (2) 不要给字典指定长度,字典的长度会在初始化或创建过程中动态增长。
- (3) Key 必须是支持比较运算(==、!=)的数据类型,比如整型、浮点型、指针、数组、结构体、接口等,而不能是函数、字典、切片这几种类型。
- (4) Value 类型可以是 Go 语言的任何基本数据类型。

例如:

```
var map1 map[string]int
```

在该例中,声明了一个键值类型为字符串,值类型为整型的字典 map1。

6.3.2 字典的初始化和创建

字典声明好后必须经过初始化或创建才能使用,未初始化或创建的字典值为 nil。可以使用“{}”操作符对字典进行初始化,或使用 make() 函数来创建字典。初始化或创建后,可以使用“=”操作符向字典动态增添数据项了。

例如,下面的操作语句编译时会出错:

```
var map1 map[string]int
map1["key1"] = 1
```

出错的原因是虽然声明了字典 map1,但 map1 没有被初始化,所以系统并没有给它分配存储空间,也就不能向 map1 中添加数据项了。

下面的操作语句是正确的:

```
var map1 map[string]int {}
map1["key1"] = 1
```

上面的操作语句正确,因为在声明字典 map1 的同时使用“{}”操作符对字典进行了初始化,也就意味着系统给 map1 分配了存储空间,当然就可以使用“=”操作符向 map1 添加数据项了。

下面的操作也是可行的:

```
var map1 map[string]int
map1 = make(map[string]int)
map1["key1"] = 1
```

这里首先声明字典 map1,然后使用 make() 函数来创建 map1,make 函数操作的实质就是给 map1 分配存储空间,所以 make 后也可以使用“=”操作符向 map1 添加数据项。

例 6-11 字典的初始化与创建。

```
1 //字典的初始化与创建
2 package main
```

```

3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     //字典声明后如果没初始化为 nil
10    var map1 map[string]int
11    fmt.Println(map1)
12    //未初始化 map1["key1"] = 1 语句编译错误
13    //使用 make 创建后就可以给字典增加数据项了
14    map1 = make(map[string]int)
15    map1["key1"] = 1
16    fmt.Println(map1)
17    //使用 "{}" 初始化后也可以给字典增加数据项
18    var map2 = map[string]int{}
19    map2["key2"] = 2
20    fmt.Println(map2)
21 }

```

编译并运行该程序,输出结果为:

```

map[]
map[key1:1]
map[key2:2]

```

在例 6-11 中,第 10 行语句声明字典 map1 且没有初始化,所以 map1 初值为 nil,不能增加数据项。第 14 行语句使用 make 函数对 map1 进行创建,然后就可以为 map1 增加数据项了。第 18 行语句在定义字典 map2 的同时,使用“{}”对其进行了初始化,所以就可以直接为 map2 增加数据项了。

6.3.3 字典的访问和操作

字典是通过 Key 来访问 Value 的,访问格式如下:

```
Value = mapName[Key]
```

字典使用键值的访问形式和数组使用下标的访问形式有些类似,其实数组可以看作是一个键值类型为整型的字典。访问以后就可以对字典中的键值对进行查找或删除操作了。

1. 字典项查找

在 Go 语言中,要从字典中查找一个特定的键值对,可以通过下面的语句来实现:

```
v, OK := mapName[Key]
```

这条语句执行后,如果查找的 Key 值存在,则将 Key 对应的 Value 值赋予 v, OK 为

true; 反之, 如果 Key 不存在, 则 v 等于 0, OK 为 false。

例 6-12 字典项的查找。

```
1 //字典项查找
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var map1 = map[string]int{"key1": 100, "key2": 200}
10    //key 值存在
11    v, OK := map1["key1"]
12    if OK {
13        fmt.Println(v, OK)
14    } else {
15        fmt.Println(v)
16    }
17    //key 值不存在
18    v, OK := map1["key3"]
19    if OK {
20        fmt.Println(v, OK)
21    } else {
22        fmt.Println(v)
23    }
24 }
```

编译并运行该程序, 输出结果为:

```
100 true
0
```

在例 6-12 中, 第 11 行语句要查找的键值“key1”存在, 所以 v=100, OK=true; 第 18 行语句要查找的键值“key3”不存在, 所以 v=0。

字典的这种查找操作, 还可以避免一个隐藏错误的发生, 那就是 Key 值可能存在, 而且 Value 值正好为 0 的情况。另外, 再配合“:=”操作符, 使得上述查找语句看起来清晰易懂。

2. 字典项删除

Go 语言提供了内置函数 delete(), 用于删除容器内的元素。delete() 函数也可以用于删除 Map 内的键值对。

例如:

```
delete(map1, "key1")
```

上面的语句将从 map1 中删除键值为“key1”的键值对, 如果“key1”这个键不存在, 那么

这个调用将什么也不会发生,也不会产生什么副作用。但是,如果传入的 Map 变量值是 nil,该调用将导致程序出现异常,这一点在使用时要特别注意。

例 6-13 字典项的删除与增添。

```
1 //字典项的删除与增添
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var map1 = map[string]int{"key1": 100, "key2": 200, "key3": 300}
10    //使用 range 遍历 map1 的过程中删除和增加字典项
11    for k, v := range map1 {
12        fmt.Println(k, v)
13        //删除字典项
14        if k == "key2" {
15            delete(map1, k)
16        }
17        //增添字典项
18        if k == "key3" {
19            map1["key4"] = 400
20        }
21    }
22    fmt.Println(map1)
23 }
```

编译并运行该程序,输出结果为:

```
key1 100
key2 200
key3 300
map[key1:100 key4:400 key3:300]
```

例 6-13 中,第 15 行语句删除键值为“key2”的字典项;第 19 行语句增加键值为“key4”的字典项。

6.4 Go 语言内存分配机制

Go 具有两种分配内存的机制,分别是使用内置函数 new() 和 make()。它们所做的事情不同,所应用到的类型也不同,这可能引起混淆,但规则却很简单。

6.4.1 new 函数

new() 函数可以给一个值类型的数据分配内存,调用成功后返回一个初始化的内存块指针,new() 函数的原型定义如下:

```
func new(Type) * Type
```

在调用 `new()` 函数时要注意,参数 `Type` 是一个类型而不是具体的数值,函数调用成功后返回该类型的内存指针,同时该类型被初始化为“0”值。

`new` 是一个分配内存的内置函数,但不同于其他语言中 `new` 所作的工作,它只是将内存清零,而不是初始化内存。比如 `new(T)` 为一个类型为 `T` 的新项目分配了值为零的存储空间并返回其地址,也就是一个类型为 `*T` 的值。用 Go 的术语来说,就是它返回了一个指向新分配的类型为 `T` 的零值的指针。

6.4.2 make 函数

`make()` 函数用于给引用类型分配内存空间,比如像 `Slice`、`Map`、`Channel` 等。需要注意的是 `make()` 函数创建的是一个引用类型对象,而不是一个内存空间的指针。`make()` 函数的原型定义如下:

```
func make(Type, size IntegerType) Type
```

在调用 `make()` 函数时,参数 `Type` 必须是一个引用类型(`Slice`、`Map` 或 `Channel`);参数 `IntegerType` 指定要创建的该对象的个数。和 `new()` 函数不同的是,`make()` 函数调用成功后返回的是一个对象,而非一个内存空间的指针,即到底要分配多少内存由该对象的大小和个数来决定。

例如,下面的代码分配了一个整型数组,长度为 10,容量为 100,并返回前 10 个数组的切片。

```
make([]int,10,100)
```

以下示例说明了 `new()` 函数和 `make()` 函数的不同。

```
var p * []int = new([]int)
var v []int = make([]int,10)
```

上例中第一条语句使用 `new()` 函数为切片结构分配内存, `*p == nil`,在实际设计中这种用法很少使用。第二条语句使用 `make()` 函数创建了一个有 10 个整数元素的数组的 `Slice` 对象。

综上所述,`make(T, args)` 函数的目的与 `new(T)` 不同。它仅用于创建 `Slice`、`Map` 和 `Channel`,并返回类型 `T` (不是 `*T`) 的一个被初始化了的 (不是零) 实例。这种差别的出现是由于这三种类型实质上是对在使用前必须进行初始化的数据结构的引用。例如, `Slice` 是一个具有三项内容的描述符,包括指向数据 (在一个数组内部) 的指针、长度以及容量,在这三项内容被初始化之前, `Slice` 值为 `nil`。对于 `Slice`、`Map` 和 `Channel`, `make()` 函数初始化了其内部的数据结构并准备了将要使用的值。

6.5 字节切片标准库

Go 语言标准库 bytes 包,提供了对字节切片进行读写操作的一系列函数和方法,包括字节切片处理函数、Buffer 对象和 Reader 对象,类似于 strings 包。

6.5.1 字节切片处理函数

Go 语言提供的字节切片函数比较多,分为基本处理函数、比较函数、后缀检查函数、索引函数、分割函数、大小写处理函数和子切片处理函数等。

1. 字节切片基本处理函数

字节切片基本处理函数主要有 Contains()、Count()、Map()、Repeat()、Replace()、Runes()和 Join()共 7 个函数,下面分别介绍。

(1) Contains() 函数。Contains() 函数的功能是检查字节切片 b 是否包含子切片 subslice,如果包含返回 true,否则返回 false。该函数原型定义如下:

```
func Contains(b, subslice []byte) bool
```

例如:

```
func main() {
    b := []byte("Golang")
    subslice1 := []byte("go")
    subslice2 := []byte("Go")
    fmt.Println(bytes.Contains(s, subslice1))
    fmt.Println(bytes.Contains(s, subslice2))
}
```

该例测试结果为:

```
false
true
```

(2) Count() 函数。Count() 函数的功能是计算子字节切片 sep 在字节切片 s 中非重叠实例的数量,该函数原型定义如下:

```
func Count(s, sep []byte) int
```

例如:

```
func main() {
    s := []byte("banana")
    sep1 := []byte("ban")
}
```

```

sep2 := []byte("na")
sep3 := []byte("a")
fmt.Println(bytes.Count(s, sep1))
fmt.Println(bytes.Count(s, sep2))
fmt.Println(bytes.Count(s, sep3))
}

```

该例测试结果为：

```

1
2
3

```

(3) Map()函数。Map()函数的功能是：首先把 s 转换为 UTF-8 编码的字节序列，然后使用 mapping 函数把 s 中的每个 Unicode 字符映射成对应的字符，最后将映射结果存放到一个新创建的字节切片中，并返回此新字节切片。该函数原型定义如下：

```
func Map(mapping func(r rune) rune, s []byte) []byte
```

在函数 Map()中，参数 mapping 映射函数；参数 s 是被处理的字节切片。

例如：

```

func main() {
    s := []byte("同学们,上午好!")
    m := func(r rune) rune {
        if r == '上' {
            r = '下'
        }
        return r
    }
    fmt.Println(string(s))
    fmt.Println(string(bytes.Map(m, s)))
}

```

该例测试结果为：

```

同学们,上午好!
同学们,下午好!

```

(4) Repeat()函数。Repeat()函数的功能是把切片 b 复制 count 次组合成一个新的字节切片并返回。该函数原型定义如下：

```
func Repeat(b []byte, count int) []byte
```

在函数 Repeat()中，参数 b 是被复制的字节切片；参数 count 是复制次数。

例如：

```
func main() {
    b := []byte("na")
    count := 2
    fmt.Println("ba" + string(bytes.Repeat(b, count)))
}
```

该例测试结果为：

```
banana
```

(5) Replace()函数。Replace()函数的功能是返回字节切片 s 的一个副本,并把前 n 个不重叠的子切片 old 替换为 new; 如果 $n < 0$ 则不限制替换的数量。该函数原型定义如下:

```
func Replace(s, old, new []byte, n int) []byte
```

在函数 Replace()中,参数 s 是原字节切片; 参数 old 是被替换子切片; 参数 new 是替换切片; 参数 n 是替换次数。

例如:

```
func main() {
    s := []byte("google")
    old := []byte("o")
    new := []byte("oo")
    n := 1
    fmt.Println(string(bytes.Replace(s, old, new, n)))
    fmt.Println(string(bytes.Replace(s, old, new, -1)))
}
```

该例测试结果为:

```
google
goooogle
```

(6) Runes()函数。Runes()函数的功能是把 s 转换为 UTF-8 编码的字节序列,并返回对应的 Unicode 切片。该函数原型定义如下:

```
func Runes(s []byte) []rune
```

在函数 Runes()中,参数 s 是需要被转换的字节切片。

例如:

```
func main() {
    s := []byte("中华人民共和国")
    r := bytes.Runes(s)
    fmt.Printf("转换前字符串 %q 长度: %d 字节\n", string(s), len(s))
    fmt.Printf("转换后字符串 %q 长度: %d 字节\n", string(r), len(r))
}
```


该例测试结果为：

转换前字符串 "中华人民共和国" 长度：21 字节

转换后字符串 "中华人民共和国" 长度：7 字节

(7) Join() 函数。Join() 函数的功能是用字节切片 sep 把 a 中的每个字节切片连接成一个字节切片并返回。该函数原型定义如下：

```
func Join(s [][]byte, sep []byte) []byte
```

在函数 Join() 中，参数 s 是字节切片的切片；参数 sep 是用于连接的字节切片。

例如：

```
func main() {  
    s := [][]byte{  
        []byte("你好"),  
        []byte("世界!")  
    }  
    sep := []byte(",")  
    fmt.Println(string(bytes.Join(s, sep)))  
}
```

该例测试结果为：

你好,世界!

2. 字节切片比较函数

字节切片比较函数共有三个：Compare()、Equal() 和 EqualFold()，它们的作用和返回值都有所区别，下面分别介绍。

(1) Compare() 函数。Compare() 函数的功能是根据字节的值比较字节切片 a 和 b 的大小，如果 $a=b$ ，返回 0；如果 $a>b$ ，返回 1；如果 $a<b$ ，返回 -1。该函数原型定义如下：

```
func Compare(a, b []byte) int
```

在函数 Compare() 中，参数 a、b 是需要比较的两个字节切片。

(2) Equal() 函数。Equal() 函数的功能是比较两个字节切片是否相等，如果参数为 nil，则等同于空的字节切片。如果 $a=b$ ，返回 true；否则，返回 false。该函数原型定义如下：

```
func Equal(a, b []byte) bool
```

在函数 Equal() 中，参数 a、b 是需要比较的两个字节切片。

(3) EqualFold() 函数。EqualFold() 函数的功能是把 s 和 t 转换成 UTF-8 字符串进行比较，并且忽略大小写。如果 $s=t$ ，返回 true；否则，返回 false。该函数原型定义如下：

```
func EqualFold(s, t []byte) bool
```

在函数 EqualFold() 中, 参数 s、t 是需要比较的两个字节切片。

例如:

```
func main() {
    a := []byte("abc")
    b := []byte("Abc")
    s := []byte("GOLANG")
    t := []byte("golang")
    fmt.Println(bytes.Compare(a, b))
    fmt.Println(bytes.Equal(a, b))
    fmt.Println(bytes.EqualFold(s, t))
}
```

该例测试结果为:

```
1
false
true
```

3. 字节切片前后缀检查函数

HasPrefix() 函数和 HasSuffix() 函数可以检查字节切片的前后缀, 并返回一个布尔型的检查结果。

(1) HasPrefix() 函数。HasPrefix() 函数的功能是检查字节切片 s 的前缀是否为 prefix, 如果是, 函数返回 true; 否则, 返回 false。该函数原型定义如下:

```
func HasPrefix(s, prefix []byte) bool
```

在函数 HasPrefix() 中, 参数 s 是需要检查的字节切片; 参数 prefix 是前缀切片。

(2) HasSuffix() 函数。HasSuffix() 函数的功能是检查字节切片 s 的后缀是否为 suffix, 如果是, 函数返回 true; 否则, 返回 false。该函数原型定义如下:

```
func HasSuffix(s, suffix []byte) bool
```

在函数 HasSuffix() 中, 参数 s 是需要检查的字节切片; 参数 suffix 是后缀切片。

例如:

```
func main() {
    s := []byte("recover")
    prefix := []byte("re")
    suffix := []byte("ed")
    fmt.Println(bytes.HasPrefix(s, prefix))
    fmt.Println(bytes.HasSuffix(s, suffix))
}
```

该例测试结果为：

```
true  
false
```

4. 字节切片位置索引函数

字节切片位置索引函数共有 8 个：Index()、IndexAny()、IndexByte()、IndexFunc()、IndexRune()、LastIndex()、LastIndexAny()和 LastIndexFunc()，下面分别介绍。

(1) Index()函数。Index()函数的功能是返回 sep 在 s 中第一次出现的位置索引(从 0 开始)，如果 sep 不在 s 中则返回 -1。该函数原型定义如下：

```
func Index(s, sep []byte) int
```

在函数 Index()中，参数 s 是主字节切片；参数 sep 是子字节切片。

(2) IndexAny()函数。IndexAny()函数的功能是把 s 解释为 UTF-8 编码的字节序列，返回 chars 中任一字符在 s 中第一次出现的位置索引；如果 s 中不包含 chars 中任何一个字符则返回 -1。该函数原型定义如下：

```
func IndexAny(s []byte, chars string) int
```

在函数 IndexAny()中，参数 s 是主字节切片；参数 chars 表示保存的 Unicode 字符集。

(3) IndexByte()函数。IndexByte()函数的功能是检查字节 c 在 s 中第一次出现的位置索引；如果 s 中不包含 c 则返回 -1。该函数原型定义如下：

```
func IndexByte(s []byte, c byte) int
```

在函数 IndexByte()中，参数 s 是主字节切片；参数 c 表示字节类型，比如 ASCII 字符。

(4) IndexFunc()函数。IndexFunc()函数的功能是把 s 解释成 UTF-8 字节序列，并返回第一个满足 f(c)=true 的字符 c 的位置索引；如果没有字符满足 f(c)=true 则返回 -1。该函数原型定义如下：

```
func IndexFunc(s []byte, f func(r rune) bool) int
```

在函数 IndexFunc()中，参数 s 是主字节切片；参数 f 表示一个函数，该函数参数是 rune 类型，返回值是 bool，如果 rune 符合 f 函数的逻辑那么返回 true，否则返回 false。

(5) IndexRune()函数。IndexRune()函数的功能是把 s 解释为 UTF-8 字节序列，并返回 rune 类型的字符 r 在 s 中的位置索引；如果 s 中不包含 r 则返回 -1。该函数原型定义如下：

```
func IndexRune(s []byte, r rune) int
```

在函数 IndexRune()中，参数 s 是主字节切片；参数 r 表示 rune 类型的字符。

(6) `LastIndex()` 函数。`LastIndex()` 函数的功能是返回 `sep` 在 `s` 中最后一次出现的位置索引, 如果 `s` 中不包含 `sep` 则返回 `-1`。该函数原型定义如下:

```
func LastIndex(s, sep []byte) int
```

在函数 `LastIndex()` 中, 参数 `s` 是主字节切片; 参数 `sep` 是子字节切片。

(7) `LastIndexAny()` 函数。`LastIndexAny()` 函数的功能是把 `s` 解释为 UTF-8 编码的字节序列, 返回 `chars` 中任一字符在 `s` 中最后出现的位置索引。如果 `chars` 为空或者 `s` 中不包含 `chars` 中的任意字符, 则返回 `-1`。该函数原型定义如下:

```
func LastIndexAny(s []byte, chars string) int
```

在函数 `LastIndexAny()` 中, 参数 `s` 是主字节切片; 参数 `chars` 表示保存的 Unicode 字符集。

(8) `LastIndexFunc()` 函数。`LastIndexFunc()` 函数的功能是把 `s` 解释为 UTF-8 编码的字节序列, 返回满足 `f(c)=true` 的字符 `c` 在 `s` 中最后一次出现的位置索引; 如果没有找到这样的字符则返回 `-1`。该函数原型定义如下:

```
func LastIndexFunc(s []byte, f func(r rune) bool) int
```

在函数 `LastIndexFunc()` 中, 参数 `s` 是主字节切片; 参数 `f` 表示一个函数, 该函数参数是 `rune` 类型, 返回值是 `bool`, 如果 `rune` 符合 `f` 函数的逻辑那么返回 `true`, 否则返回 `false`。

例如:

```
func main() {
    s := []byte("Google")
    sep := []byte("o")
    chars := "gle"
    var c byte = 'g'
    var r rune = 'l'
    fmt.Printf("切片 %q 在 %q 中的位置索引是: %d\n",
        sep, s, bytes.Index(s, sep))
    fmt.Printf("切片 %q 中的任一字符在 %q 中的位置索引是: %d\n",
        chars, s, bytes.IndexAny(s, chars))
    fmt.Printf("byte 型字符 %q 在 %q 中的位置索引是: %d\n",
        c, s, bytes.IndexByte(s, c))
    fmt.Printf("切片 %q 第一个符合 f() 的字符索引是: %d\n",
        s, bytes.IndexFunc(s, f))
    fmt.Printf("rune 型字符 %q 在 %q 中的位置索引是: %d\n",
        r, s, bytes.IndexRune(s, r))
    fmt.Printf("切片 %q 在 %q 中的最后位置索引是: %d\n",
        sep, s, bytes.LastIndex(s, sep))
    fmt.Printf("切片 %q 中的任一字符在 %q 中的最后位置索引是: %d\n",
        chars, s, bytes.LastIndexAny(s, chars))
    fmt.Printf("切片 %q 最后一个符合 f() 的字符索引是: %d\n",
```

```

        s, bytes.LastIndexFunc(s, f))
    }
    func f(a rune) bool {
        if a > 'k' {
            return true
        } else {
            return false
        }
    }
}

```

该例测试结果为：

切片"o"在"Google"中的位置索引是：1
 切片"gle"中的任一字符在"Google"中的位置索引是：3
 byte 型字符'g'在"Google"中的位置索引是：3
 切片"Google"第一个符合 f() 的字符索引是：1
 rune 型字符'l'在"Google"中的位置索引是：4
 切片"o"在"Google"中的最后位置索引是：2
 切片"gle"中的任一字符在"Google"中的最后位置索引是：5
 切片"Google"最后一个符合 f() 的字符索引是：4

5. 字节切片分割函数

字节切片分割函数共有 6 个：Fields()、FieldsFunc()、Split()、SplitN()、SplitAfter() 和 SplitAfterN()，下面分别介绍。

(1) Fields() 函数。Fields() 函数的功能是把字节切片 s 按照一个或者连续多个空白字符分割成多个字节切片，如果 s 只包含空白字符则返回空字节切片。该函数原型定义如下：

```
func Fields(s []byte) [][]byte
```

在函数 Fields() 中，参数 s 是准备分割的字节切片。

例如：

```

func main() {
    s := []byte("I'm a student.")
    for _, f := range bytes.Fields(s) {
        fmt.Printf("%q ", f)
    }
}

```

该例测试结果为：

```
"I'm" "a" "student."
```

(2) FieldsFunc() 函数。FieldsFunc() 函数的功能是把 s 解释为 UTF-8 编码的字符序列，对于每个 Unicode 字符 c，如果 f(c) 返回 true 就把 c 作为分割字符对 s 进行拆分。如果

所有字符都满足 $f(c)$ 为 `true`, 则返回空的切片。该函数原型定义如下:

```
func FieldsFunc(s []byte, f func(rune) bool) [][]byte
```

在函数 `FieldsFunc()` 中, 参数 `s` 是准备分割的字节切片; 参数 `f` 表示一个函数, 该函数参数是 `rune` 类型, 返回值是 `bool`, 如果 `rune` 符合 `f` 函数的逻辑那么返回 `true`, 否则返回 `false`。

例如:

```
func main() {
    s := []byte("小明和爸爸和妈妈")
    for _, f := range bytes.FieldsFunc(s, f) {
        fmt.Printf("%q", f)
    }
}

func f(a rune) bool {
    if a == '和' {
        return true
    } else {
        return false
    }
}
```

该例测试结果为:

"小明" "爸爸" "妈妈"

(3) `Split()` 函数。`Split()` 函数的功能是把 `s` 用 `sep` 分割成多个字节切片返回。如果 `sep` 为空, `Split` 则把 `s` 切分成每个字节切片对应一个 UTF-8 字符。`Split` 等效于参数 `n` 为 -1 的 `SplitN` 函数。该函数原型定义如下:

```
func Split(s, sep []byte) [][]byte
```

在函数 `Split()` 中, 参数 `s` 是准备分割的字节切片; 参数 `sep` 是用于分割的子切片。

例如:

```
func main() {
    s := []byte("Hello, world!")
    sep := []byte(",")
    for _, f := range bytes.Split(s, sep) {
        fmt.Printf("%q", f)
    }
}
```

该例测试结果为:

"Hello" "world!"

(4) SplitN()函数。SplitN()函数的功能是把 s 用 sep 分割成多个字节切片并返回。如果 sep 为空, Split 则把 s 切分成每个字节切片对应一个 UTF-8 字符。参数 n 决定返回的切片的长度: 如果 $n > 0$, 最多返回 n 个子切片, 子切片可能包含未切分的字节序列; 如果 $n = 0$, 返回空切片; 如果 $n < 0$, 返回所有的子切片。该函数原型定义如下:

```
func SplitN(s, sep []byte, n int) [][]byte
```

在函数 SplitN() 中, 参数 s 是准备分割的字节切片; 参数 sep 是用于分割的子切片; 参数 n 表示分割切片的长度。

例如:

```
func main() {
    s := []byte("南瓜, 黄瓜, 西红柿, 茄子")
    sep := []byte(",")
    n := 2
    for _, f := range bytes.SplitN(s, sep, n) {
        fmt.Printf("%q ", f)
    }
    fmt.Println()
    for _, f := range bytes.SplitN(s, sep, 0) {
        fmt.Printf("%q ", f)
    }
    fmt.Println()
    for _, f := range bytes.SplitN(s, sep, -1) {
        fmt.Printf("%q ", f)
    }
}
```

该例测试结果为:

```
"南瓜" "黄瓜, 西红柿, 茄子"
nil
"南瓜" "黄瓜" "西红柿" "茄子"
```

(5) SplitAfter()函数。SplitAfter()函数的功能是用 sep 作为后缀把 s 切分成多个字节切片并返回。如果 sep 为空, 则把 s 切分成每个字节切片对应一个 UTF-8 字符。该函数原型定义如下:

```
func SplitAfter(s, sep []byte) [][]byte
```

在函数 SplitAfter() 中, 参数 s 是准备分割的字节切片; 参数 sep 是用于分割的后缀。

例如:

```
func main() {
    s := []byte("managerteacherworkerfarmerstudent")
    sep := []byte("er")
    for _, f := range bytes.SplitAfter(s, sep) {
```

```

        fmt.Printf("%q", f)
    }
}

```

该例测试结果为：

```
"manager" "teacher" "worker" "farmer" "student"
```

(6) SplitAfterN()函数。SplitAfterN()函数的功能是用 sep 作为后缀把 s 切分成多个字节切片并返回。如果 sep 为空,则把 s 切分成每个字节切片对应一个 UTF-8 字符。参数 n 决定返回的切片的长度: 如果 $n > 0$, 最多返回 n 个子切片, 子切片可能包含未切分的字节序列; 如果 $n = 0$, 返回空切片; 如果 $n < 0$, 返回所有的子切片。该函数原型定义如下:

```
func SplitAfterN(s, sep []byte, n int) [][]byte
```

在函数 SplitAfterN()中, 参数 s 是准备分割的字节切片; 参数 sep 是用于分割的后缀; 参数 n 表示分割切片的长度。

例如:

```

func main() {
    s := []byte("managerteacherworkerfarmerstudent")
    sep := []byte("er")
    n := 3
    for _, f := range bytes.SplitAfterN(s, sep, n) {
        fmt.Printf("%q", f)
    }
    fmt.Println()
    for _, f := range bytes.SplitAfterN(s, sep, 0) {
        fmt.Printf("%q", f)
    }
    fmt.Println()
    for _, f := range bytes.SplitAfterN(s, sep, -1) {
        fmt.Printf("%q", f)
    }
}

```

该例测试结果为:

```

"manager" "teacher" "workerfarmerstudent"
nil
"manager" "teacher" "worker" "farmer" "student"

```

6. 字节切片大小写处理函数

字节切片大小写处理函数共有 7 个: Title()、ToTitle()、ToTitleSpecial()、ToLower()、ToLowerSpecial()、ToUpper()和 ToUpperSpecial(), 下面分别介绍。

(1) Title()函数。Title()函数的功能是返回 s 的一个副本,把 s 中每个单词的首字母改为 Unicode 字符的大写。该函数原型定义如下:

```
func Title(s []byte) []byte
```

在函数 Title()中,参数 s 是需要被修改的字节切片。

(2) ToTitle()函数。ToTitle()函数的功能是返回 s 的一个副本,并把其中所有的 Unicode 字符转换为大写。该函数原型定义如下:

```
func ToTitle(s []byte) []byte
```

在函数 ToTitle()中,参数 s 是需要被修改的字节切片。

(3) ToTitleSpecial()函数。ToTitleSpecial()函数的功能是返回 s 的一个副本,并把其中的所有 Unicode 字符都根据_case 指定的规则转换成大写。该函数原型定义如下:

```
func ToTitleSpecial(_case unicode.SpecialCase, s []byte) []byte
```

在函数 ToTitleSpecial()中,参数 s 是需要被修改的字节切片;参数_case 是转换规则。

例如:

```
func main() {  
    s := []byte("hello, world!")  
    fmt.Println(string(bytes.Title(s)))  
    fmt.Println(string(bytes.ToTitle(s)))  
    fmt.Println(string(bytes.ToTitleSpecial(unicode.AzeriCase, s)))  
}
```

该例测试结果为:

```
Hello, World!  
HELLO, WORLD!  
HELLO, WORLD!
```

(4) ToLower()函数。ToLower()函数的功能是返回 s 的一个副本,并把其中所有的 Unicode 字符转换为小写。该函数原型定义如下:

```
func ToLower(s []byte) []byte
```

在函数 ToLower()中,参数 s 是需要被修改的字节切片。

(5) ToLowerSpecial()函数。ToLowerSpecial()函数的功能是返回 s 的一个副本,并把其中的所有 Unicode 字符都根据_case 指定的规则转换成小写。该函数原型定义如下:

```
func ToLowerSpecial(_case unicode.SpecialCase, s []byte) []byte
```

在函数 ToLowerSpecial()中,参数 s 是需要被修改的字节切片;参数_case 是转换规则

(可参见 unicode 包)。

例如：

```
func main() {  
    s := []byte("Hello, World!")  
    fmt.Println(string(bytes.ToLower(s)))  
    fmt.Println(string(bytes.ToLowerSpecial(unicode.AzeriCase, s)))  
}
```

该例测试结果为：

```
hello, world!  
hello, world!
```

(6) ToUpper()函数。ToUpper()函数的功能是返回 s 的一个副本,并把其中所有的 Unicode 字符转换为大写。该函数原型定义如下：

```
func ToUpper(s []byte) []byte
```

在函数 ToUpper()中,参数 s 是需要被修改的字节切片。

(7) ToUpperSpecial()函数。ToUpperSpecial()函数的功能是返回 s 的一个副本,并把其中的所有 Unicode 字符都根据_case 指定的规则转换成大写。该函数原型定义如下：

```
func ToUpperSpecial(_case unicode.SpecialCase, s []byte) []byte
```

在函数 ToUpperSpecial()中,参数 s 是需要被修改的字节切片;参数_case 是转换规则。

例如：

```
func main() {  
    s := []byte("hello, world!")  
    fmt.Println(string(bytes.ToUpper(s)))  
    fmt.Println(string(bytes.ToUpperSpecial(unicode.AzeriCase, s)))  
}
```

该例测试结果为：

```
HELLO, WORLD!  
HELLO, WORLD!
```

7. 子字节切片处理函数

子字节切片处理函数共有 7 个: Trim()、TrimFunc()、TrimLeft()、TrimLeftFunc()、TrimRight()、TrimRightFunc()和 TrimSpace();下面分别介绍。

(1) Trim()函数。Trim()函数的功能是返回 s 的子字节切片, cutset 中任意出现在 s 的首部和尾部的连续字符将被删除。该函数原型定义如下：

```
func Trim(s []byte, cutset string) []byte
```

在函数 Trim() 中, 参数 s 是需要被处理的字节切片; 参数 cutset 是参照字符串。

例如:

```
func main() {
    s := []byte("hello, world!")
    cutset := "hold!"
    fmt.Println(string(bytes.Trim(s, cutset)))
}
```

该例测试结果为:

```
ello, wor
```

分析上例执行结果, 因为“hold!”中的“h”出现在 s 的首部, “ld!”出现在 s 的尾部, 将被删除。所以, 执行结果为“ello, wor”。

(2) TrimFunc() 函数。TrimFunc() 函数的功能是返回 s 的子字节切片, 不包含 s 首部和尾部连接的满足 $f(c)=true$ 的字符 c。该函数原型定义如下:

```
func TrimFunc(s []byte, f func(r rune) bool) []byte
```

在函数 TrimFunc() 中, 参数 s 是需要被处理的字节切片; 参数 f 表示一个函数, 该函数参数是 rune 类型, 返回值是 bool, 如果 rune 符合 f 函数的逻辑那么返回 true, 否则返回 false。

例如:

```
func main() {
    s := []byte("hello, world!")
    fmt.Println(string(bytes.TrimFunc(s, f)))
}

func f(a rune) bool {
    if a == 'h' || a == 'l' {
        return true
    } else {
        return false
    }
}
```

该例测试结果为:

```
ello, world
```

(3) TrimLeft() 函数。TrimLeft() 函数的功能是返回 s 的子字节切片, cutset 中任意出现在 s 首部的连续字符将被删除。该函数原型定义如下:

```
func TrimLeft(s []byte, cutset string) []byte
```

在函数 TrimLeft() 中, 参数 s 是需要被处理的字节切片; 参数 cutset 是参照字符串。

```
func main() {
    s := []byte("hello, world!")
    cutset := "hold"
    fmt.Println(string(bytes.TrimLeft(s, cutset)))
}
```

该例测试结果为:

```
ello, world!
```

(4) TrimLeftFunc() 函数。TrimLeftFunc() 函数的功能是返回 s 的一个子字节切片, 不包含 s 首部连续满足 f(c)=true 的字符 c。该函数原型定义如下:

```
func TrimLeftFunc(s []byte, f func(r rune) bool) []byte
```

在函数 TrimLeftFunc() 中, 参数 s 是需要被处理的字节切片; 参数 f 表示一个函数, 该函数参数是 rune 类型, 返回值是 bool, 如果 rune 符合 f 函数的逻辑那么返回 true, 否则返回 false。

例如:

```
func main() {
    s := []byte("hello, world!")
    fmt.Println(string(bytes.TrimLeftFunc(s, f)))
}

func f(a rune) bool {
    if a == 'h' || a == 'H' {
        return true
    } else {
        return false
    }
}
```

该例测试结果为:

```
ello, world!
```

(5) TrimRight() 函数。TrimRight() 函数的功能是返回 s 的子字节切片, cutset 中任意出现在 s 尾部的连续字符将被删除。该函数原型定义如下:

```
func TrimRight(s []byte, cutset string) []byte
```

在函数 TrimRight() 中, 参数 s 是需要被处理的字节切片; 参数 cutset 是参照字符串。

例如：

```
func main() {  
    s := []byte("hello, world!")  
    cutset := "hold!"  
    fmt.Println(string(bytes.TrimRight(s, cutset)))  
}
```

该例测试结果为：

```
hello, wor
```

(6) TrimRightFunc()函数。TrimRightFunc()函数的功能是返回 s 的一个子字节切片,不包含 s 尾部连续满足 f(c)=true 的字符 c。该函数原型定义如下：

```
func TrimRightFunc(s []byte, f func(r rune) bool) []byte
```

在函数 TrimRightFunc()中,参数 s 是需要被处理的字节切片;参数 f 表示一个函数,该函数参数是 rune 类型,返回值是 bool,如果 rune 符合 f 函数的逻辑那么返回 true,否则返回 false。

例如：

```
func main() {  
    s := []byte("hello, world!")  
    fmt.Println(string(bytes.TrimRightFunc(s, f)))  
}  
func f(a rune) bool {  
    if a == '!' || a == '.' {  
        return true  
    } else {  
        return false  
    }  
}
```

该例测试结果为：

```
hello, world
```

(7) TrimSpace()函数。TrimSpace()函数的功能是返回 s 的一个子字节切片,并删除 s 中开始和结尾处的连续的 Unicode 空白字符。该函数原型定义如下：

```
func TrimSpace(s []byte) []byte
```

在函数 TrimSpace()中,参数 s 是需要被处理的字节切片。

例如：

```
func main() {  
    s := []byte(" hello,world! ")  
    fmt.Println(string(bytes.TrimSpace(s)))  
}
```

该例测试结果为：

```
hello ,world!
```

6.5.2 Buffer 创建函数及操作方法

在 bytes 包中,缓冲区(Buffer)是最常用的对字节切片进行 I/O 操作的对象(关于对象的概念可参见第 8 章内容)。Buffer 对象定义如下:

```
type Buffer struct {  
    //contains filtered or unexported fields  
}
```

默认情况下 Buffer 对象没有定义初始值,Buffer 使用结构体自带的一个 [64]byte 数组作为存储空间。当超出限制时,另创建一个两倍的存储空间,并复制未读取的数据。当 Buffer 里的数据被完全读取后,会将写入位置重置到底层数据的开始处。因此只要读写操作平衡,就无须担心内存会持续增长。

1. Buffer 创建函数

可以使用 NewBuffer() 和 NewBufferString() 函数创建 Buffer 对象,它们分别使用字节切片和字符串对 Buffer 进行初始化。

(1) NewBuffer() 函数。NewBuffer() 函数的功能是创建一个 Buffer,并使用字节切片 buf 对它进行初始化。buf 可以直接用来作为准备要读的数据;也可以用来指定写缓冲区的大小,这时要提前为 buf 分配内存空间,但是 len(buf) 为 0。该函数原型定义如下:

```
func NewBuffer(buf []byte) * Buffer
```

(2) NewBufferString() 函数。NewBufferString() 函数的功能是创建一个 Buffer,并用字符串 s 对它进行初始化。它通常用于读取已经存在的数据。该函数原型定义如下:

```
func NewBufferString(s string) * Buffer
```

2. Buffer 写操作方法

对 Buffer 对象的写操作方法共有 5 个: Write()、WriteByte()、WriteRune()、WriteString() 和 WriteTo() 方法,下面分别介绍。

(1) Write() 方法。Write() 方法的功能是把字节切片 p 写入 Buffer 中,该方法执行结

束返回成功写入的字节数 n 和错误类型 err 。如果返回的 n 满足 $n = \text{len}(p)$, 则 err 总是为 nil ; 如果数据太大则 `Write` 会 panic 并产生 `ErrTooLarge` 异常。该方法原型定义如下:

```
func (b * Buffer) Write(p []byte) (n int, err error)
```

在方法 `Write()` 中, 参数 p 是要写入的字节切片。

例如:

```
func main() {  
    p := []byte("Hello, world!")  
    b := bytes.NewBuffer(nil)  
    n, err := b.Write(p)  
    fmt.Println(string(b.Bytes()), n, err)  
}
```

该例测试结果为:

```
Hello, world! 12 <nil>
```

(2) `WriteByte()` 方法。`WriteByte()` 方法的功能是写入一个字节, 总是返回 nil 。这个返回值只是为了匹配 `bufio. Writer` 的 `Write` 函数。如果内部数据太大, `WriteByte` 会 panic 并产生 `ErrTooLarge` 异常。该方法原型定义如下:

```
func (b * Buffer) WriteByte(c byte) error
```

在方法 `WriteByte()` 中, 参数 c 是要写入的字节数据, 比如 ASCII 字符。

例如:

```
func main() {  
    var c1, c2 byte = 'G', 'o'  
    b := bytes.NewBuffer(nil)  
    err := b.WriteByte(c1)  
    b.WriteByte(c2)  
    fmt.Println(string(b.Bytes()), err)  
}
```

该例测试结果为:

```
Go <nil>
```

(3) `WriteRune()` 方法。`WriteRune()` 方法的功能是把一个 Unicode 字符的 UTF-8 编码写入 Buffer, 并返回写入字节数 n 。 err 总是 nil , 这个返回类型是为了和 `bufio. Writer` 的 `WriteRune` 函数匹配。如果 Buffer 的缓冲区太大, 则 `WriteRune` 会 panic 并产生 `ErrTooLarge` 异常。该方法原型定义如下:

```
func (b * Buffer) WriteRune(r rune) (n int, err error)
```


在方法 `WriteRune()` 中, 参数 `r` 是要写入的 `rune` 类型字符。

例如:

```
func main() {
    var r1, r2 rune = '中', '国'
    b := bytes.NewBuffer(nil)
    b.WriteRune(r1)
    b.WriteRune(r2)
    fmt.Println(string(b.Bytes()))
}
```

该例测试结果为:

中国

(4) `WriteString()` 方法。`WriteString()` 方法的功能是把 `s` 写入 `Buffer`, 返回值 `n` 为 `len(s)`, `err` 总是为 `nil`。如果内部缓冲区太大, `WriteString` 会 `panic` 并产生 `ErrTooLarge` 异常。该方法原型定义如下:

```
func (b * Buffer) WriteString(s string) (n int, err error)
```

在方法 `WriteString()` 中, 参数 `s` 是要写入的字符串。

例如:

```
func main() {
    s := "你好, 世界!"
    b := bytes.NewBuffer(nil)
    n, err := b.WriteString(s)
    fmt.Println(string(b.Bytes()), n, err)
}
```

该例测试结果为:

你好, 世界! 18 <nil>

(5) `WriteTo()` 方法。`WriteTo()` 方法的功能是把 `Buffer` 中的数据写入到 I/O 对象 `w` 中, 直到数据为空或者遇到错误, 返回值 `n` 总是足够用 `int` 表示, 使用 `int64` 类型是为了和 `io.WriterTo` 接口匹配。任何写入时遇到的错误都会被返回。该方法原型定义如下:

```
func (b * Buffer) WriteTo(w io.Writer) (n int64, err error)
```

在方法 `WriteTo()` 中, 参数 `w` 是要写入的 I/O 对象。

例如:

```
func main() {
    buf := []byte("Golang")
```

```

b := bytes.NewBuffer(buf)
w := bytes.NewBuffer(nil)
n, err := b.WriteTo(w)
fmt.Println(string(w.Bytes()), n, err)
}

```

该例测试结果为：

```
Golang 6 <nil>
```

3. Buffer 读操作方法

对 Buffer 的读操作方法共有 6 个：Read()、ReadByte()、ReadBytes()、ReadFrom()、ReadRune()和 ReadString()方法，下面分别介绍。

(1) Read()方法。Read()方法的功能是从 Buffer 中读取 len(p)个字节，并复制到 p 中；如果 Buffer 中未读数据不足 len(p)，则把所有的数据都复制到 p 中。该方法执行完，返回实际读取的字节数 n 和错误类型 err。如果 Buffer 中没有数据，则 err 为 io.EOF(除非 len(p)为 0)；否则 err 为 nil。该方法原型定义如下：

```
func (b * Buffer) Read(p []byte) (n int, err error)
```

在方法 Read()中，参数 p 是用来存放读取结果的字节切片。

例如：

```

func main() {
    buf := []byte("Hello, world!")
    b := bytes.NewBuffer(buf)
    var p [8]byte
    n, err := b.Read(p[:])
    fmt.Println(string(p[:n]), n, err)
}

```

该例测试结果为：

```
Hello, wo 8 <nil>
```

(2) ReadByte()方法。ReadByte()方法的功能是从 Buffer 中读取一个字节并返回，如果没有数据可读则 err 为 io.EOF。该方法原型定义如下：

```
func (b * Buffer) ReadByte() (c byte, err error)
```

例如：

```

func main() {
    buf := []byte("Golang")
}

```

```

b := bytes.NewBuffer(buf)
c, err := b.ReadByte()
fmt.Println(string(c), err)
}

```

该例测试结果为：

```
G <nil>
```

(3) ReadBytes()方法。ReadBytes()方法的功能是从 Buffer 中读取数据，直到第一次遇到分隔符“delim”，把已读取的数据包括“delim”作为字节切片返回。如果在读取到“delim”前出现错误，则返回已读取的数据和那个错误（通常是 io.EOF）。只有返回的数据不以“delim”结尾时，ReadBytes 才返回 err 非空。该方法原型定义如下：

```
func (b * Buffer) ReadBytes(delim byte) (line []byte, err error)
```

在方法 ReadBytes()中，参数 delim 是指定的分隔符。

例如：

```

func main() {
    buf := []byte("Hello, world!")
    b := bytes.NewBuffer(buf)
    var delim byte = ','
    line, err := b.ReadBytes(delim)
    fmt.Println(string(line), err)
}

```

该例测试结果为：

```
Hello, <nil>
```

(4) ReadFrom()方法。ReadFrom()方法的功能是从 I/O 接口对象 r 中读取数据，并写入到 Buffer 中，直到 r.Read 返回 io.EOF，除了 io.EOF 之外的错误会被 ReadFrom 返回。返回值 n 为已读取的字节数，err 为 r.Read 返回的错误。如果读取的数据太大，ReadFrom 会 panic 并产生 ErrTooLarge 异常。该方法原型定义如下：

```
func (b * Buffer) ReadFrom(r io.Reader) (n int64, err error)
```

在方法 ReadFrom()中，参数 r 是所要访问的 I/O 接口对象。

例如：

```

func main() {
    buf := []byte("中华人民共和国")
    r := bytes.NewBuffer(buf)
    b := bytes.NewBuffer(nil)
    n, err := b.ReadFrom(r)
    fmt.Println(string(b.Bytes()), n, err)
}

```


该例测试结果为：

```
中华人民共和国 21 <nil>
```

(5) `ReadRune()` 方法。`ReadRune()` 方法的功能是从 `Buffer` 中读取一个 `Unicode` 字符,同时返回它的 `UTF-8` 编码的字节数;如果没有数据可读则返回 `io.EOF`。如果读取的不是 `UTF-8` 编码的字节序列,则读取一个字节并返回 `U+FFFD`。该方法原型定义如下:

```
func (b * Buffer) ReadRune() (r rune, size int, err error)
```

例如:

```
func main() {  
    buf := []byte("中华人民共和国")  
    b := bytes.NewBuffer(buf)  
    r, size, err := b.ReadRune()  
    fmt.Println(string(r), size, err)  
}
```

该例测试结果为:

```
中 3 <nil>
```

(6) `ReadString()` 方法。`ReadString()` 方法的功能是读取数据直到遇到分隔符“`delim`”,并把已读取的数据包含“`delim`”作为 `string` 返回。如果在遇到“`delim`”前出错,则把已读数据作为 `string` 和遇到的错误(通常是 `io.EOF`)返回。当返回的 `string` 不以“`delim`”结尾时,才会返回非空 `err`。该方法原型定义如下:

```
func (b * Buffer) ReadString(delim byte) (line string, err error)
```

在方法 `ReadString()` 中,参数 `delim` 是指定的分隔符。

例如:

```
func main() {  
    buf := []byte("Golang is a beautiful language,I like it!")  
    b := bytes.NewBuffer(buf)  
    var delim byte = ','  
    line, err := b.ReadString(delim)  
    fmt.Println(line, err)  
}
```

该例测试结果为:

```
Golang is a beautiful language,<nil>
```

4. 其他操作方法

对 Buffer 对象的其他操作方法共有 8 个: Bytes()、Len()、Next()、Reset()、String()、Truncate()、UnreadByte()和 UnreadRune()方法,下面分别介绍。

(1) Bytes()方法。Bytes()方法的功能是返回 Buffer 中未读数据的字节切片,满足 `len(b.Bytes())=b.Len()`。如果调用者修改了返回字节切片的内容,Buffer 中的未读数据也会被修改。该方法原型定义如下:

```
func (b * Buffer) Bytes() []byte
```

(2) Len()方法。Len()方法的功能是返回 Buffer 中未读数据的字节数。它满足条件 `b.Len()=len(b.Bytes())`。该方法原型定义如下:

```
func (b * Buffer) Len() int
```

例如:

```
func main() {
    buf := []byte("Golang is a beautiful language, I like it!")
    b := bytes.NewBuffer(buf)
    var delim byte = ','
    fmt.Println("未执行读操作前 Buffer 中数据和字节数")
    fmt.Println(string(b.Bytes()), b.Len())
    b.ReadString(delim)
    fmt.Println("执行读操作后 Buffer 中数据和字节数")
    fmt.Println(string(b.Bytes()), b.Len())
}
```

该例测试结果为:

```
未执行读操作前 Buffer 中数据和字节数
Golang is a beautiful language, I like it! 41
执行读操作后 Buffer 中数据和字节数
I like it! 10
```

(3) Next()方法。Next()方法的功能是返回 Buffer 中下 n 个未读取的字节,本次读取会修改 Buffer 中的读取位置。如果 Buffer 中未读数据不足 n 字节,则返回全部未读数据。返回的字节切片在下次读或者写前有效。该方法原型定义如下:

```
func (b * Buffer) Next(n int) []byte
```

在方法 Next()中,参数 n 是要读取的字节数。

例如:

```
func main() {
    buf := []byte("Golang")
```

```
b := bytes.NewBuffer(buf)
fmt.Println(string(b.Next(4)))
fmt.Println(string(b.Next(4)))
}
```

该例测试结果为：

```
Gola
ng
```

(4) Reset()方法。Reset()方法的功能是把 Buffer 中的数据清空,等同于 b.Truncate(0)。该方法原型定义如下:

```
func (b *Buffer) Reset()
```

(5) String()方法。String()方法的功能是把 Buffer 中的未读数据作为 string 返回。如果 Buffer 是个 nil 指针,则返回“<nil>”。该方法原型定义如下:

```
func (b *Buffer) String() string
```

(6) Truncate()方法。Truncate()方法的功能是只保留前 n 个字节的数据并清除其余数据。当 $n < 0$ 或者 $n > b.Len()$ 时,则 Truncate 导致 panic; 当 $n = 0$ 时,等同于 b.Reset()。该方法原型定义如下:

```
func (b *Buffer) Truncate(n int)
```

在方法 Truncate()中,参数 n 是需要保留的字节数。

例如:

```
func main() {
    buf := []byte("Hello,world!")
    b := bytes.NewBuffer(buf)
    fmt.Println("未执行 Reset 前 Buffer 中内容:", string(b.Bytes()))
    b.Truncate(5)
    fmt.Println("执行 Truncate 后 Buffer 中内容:", string(b.Bytes()))
    b.Reset()
    fmt.Println("执行 Reset 后 Buffer 中内容:", string(b.Bytes()))
}
```

该例测试结果为:

```
未执行 Reset 前 Buffer 中内容: Hello,world!
执行 Truncate 后 Buffer 中内容: Hello
执行 Reset 后 Buffer 中内容: ""
```

(7) UnreadByte()方法。UnreadByte()方法的功能是取消上次读取的最后一个字节。

如果上次读取之后有过写操作,则返回错误。该方法原型定义如下:

```
func (b * Buffer) UnreadByte() error
```

例如:

```
func main() {  
    buf := []byte("Golang")  
    b := bytes.NewBuffer(buf)  
    fmt.Println(string(b.Next(3)))  
    b.UnreadByte()  
    fmt.Println(string(b.Bytes()))  
}
```

该例测试结果为:

```
Gol  
lang
```

(8) `UnreadRune()` 方法。`UnreadRune()` 方法的功能是取消上次 `ReadRune` 读取的 Unicode 字符。如果最近一次读写操作不是 `ReadRune`, 则 `UnreadRune` 返回一个错误。该方法原型定义如下:

```
func (b * Buffer) UnreadRune() error
```

例如:

```
func main() {  
    buf := []byte("中华人民共和国")  
    b := bytes.NewBuffer(buf)  
    b.ReadRune()  
    b.ReadRune()  
    b.ReadRune()  
    b.UnreadRune()  
    fmt.Println(string(b.Bytes()))  
}
```

该例测试结果为:

```
人民共和国
```

6.5.3 Reader 对象及方法

`Reader` 是另外一个可以对字节切片进行操作的对象,与 `Buffer` 对象不同的是, `Reader` 对象只能从字节切片中读取数据,不能写入数据。但是, `Reader` 对象支持对字节切片进行定位操作。 `Reader` 对象定义如下:

```
type Reader struct {
    //contains filtered or unexported fields
}
```

1. Reader 对象创建函数

Reader 对象的创建函数是 `NewReader()`，它的作用是创建一个 Reader，数据为字节切片 `b`。该函数原型定义如下：

```
func NewReader(b []byte) * Reader
```

2. Reader 对象操作方法

Reader 对象的操作方法共有 8 个：`Len()`、`Read()`、`ReadAt()`、`ReadByte()`、`ReadRune()`、`Seek()`、`UnreadByte()` 和 `UnreadRune()` 方法，下面分别介绍。

(1) `Len()` 方法。`Len()` 方法的作用是返回 Reader 中未读数据的字节数。该方法原型定义如下：

```
func (r * Reader) Len() int
```

例如：

```
func main() {
    b := []byte("Golang")
    r := bytes.NewReader(b)
    fmt.Println(r.Len())
}
```

该例测试结果为：

```
6
```

(2) `Read()` 方法。`Read()` 方法的作用是从 Reader 中读取 `len(b)` 个字节，返回已读取的字节数，如果遇到错误则返回错误（通常是 `io.EOF`）。该方法原型定义如下：

```
func (r * Reader) Read(b []byte) (n int, err error)
```

```
func main() {
    b := []byte("Golang")
    var buf [3]byte
    r := bytes.NewReader(b)
    n, err := r.Read(buf[:])
    fmt.Println(string(buf[:n]), n, err)
    n, err = r.Read(buf[:])
    fmt.Println(string(buf[:n]), n, err)
    n, err = r.Read(buf[:])
    fmt.Println(string(buf[:n]), n, err)
}
```

该例测试结果为：

```
Gol 3 <nil>
ang 3 <nil>
0 EOF
```

(3) ReadAt()方法。ReadAt()方法的作用是从 Reader 中偏移为 off 字节的位置读取 len(b)个字节,并返回已读取的字节数;如果遇到错误则返回错误(通常是 io.EOF)。ReadAt 读取的数据不会从 Reader 中清除。该方法原型定义如下:

```
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
func main() {
    b := []byte("Golang")
    var buf [3]byte
    r := bytes.NewReader(b)
    n, err := r.ReadAt(buf[:], 2)
    fmt.Println(string(buf[:n]), n, err)
    n, err = r.ReadAt(buf[:], 3)
    fmt.Println(string(buf[:n]), n, err)
    n, err = r.ReadAt(buf[:], 4)
    fmt.Println(string(buf[:n]), n, err)
}
```

该例测试结果为：

```
lan 3 <nil>
ang 3 <nil>
ng 2 EOF
```

(4) ReadByte()方法。ReadByte()方法的作用是从 Reader 中读取一个字节并返回,如果遇到错误则返回错误(通常是 io.EOF)。该方法原型定义如下:

```
func (r *Reader) ReadByte() (b byte, err error)
```

例如:

```
func main() {
    b := []byte("Golang")
    r := bytes.NewReader(b)
    c, err := r.ReadByte()
    fmt.Println(string(c), err)
}
```

该例测试结果为:

```
G<nil>
```

(5) ReadRune()方法。ReadRune()方法的作用是从 Reader 中按照 UTF-8 编码读取

一个 Unicode 字符,返回此字符,其字符的 UTF-8 编码占用的字节数,如果遇到错误则返回错误(通常是 io.EOF)。该方法原型定义如下:

```
func (r *Reader) ReadRune() (ch rune, size int, err error)
```

例如:

```
func main() {  
    b := []byte("中华人民共和国")  
    r := bytes.NewReader(b)  
    ch, size, err := r.ReadRune()  
    fmt.Println(string(ch), size, err)  
}
```

该例测试结果为:

```
中 3 <nil>
```

(6) Seek()方法。Seek()方法的作用是实现了 io.Seeker 接口方法,用于设置下次读或写操作的位置,返回新的偏移位置字节数和错误(如果有)。当 whence=0,从起始位置计算 offset;当 whence=1,从当前位置计算 offset;当 whence=2,从尾部位置计算 offset。该方法原型定义如下:

```
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

```
func main() {  
    b := []byte("中华人民共和国")  
    r := bytes.NewReader(b)  
    r.Seek(3, 0) //从起始位置偏移 3, 应该是'华'字。  
    ch, size, err := r.ReadRune()  
    fmt.Println(string(ch), size, err)  
    r.Seek(3, 1) //从当前位置偏移 3, 应该是'民'字。  
    ch, size, err = r.ReadRune()  
    fmt.Println(string(ch), size, err)  
    r.Seek(-6, 2) //从尾部偏移 -6, 应该是'和'字。  
    ch, size, err = r.ReadRune()  
    fmt.Println(string(ch), size, err)  
}
```

该例测试结果为:

```
华 3 <nil>  
民 3 <nil>  
和 3 <nil>
```

(7) UnreadByte()方法。UnreadByte()方法的作用是取消上次读取的最后一个字节,如果 Buffer 没有被读取过则 UnreadByte 也返回一个错误。该方法原型定义如下:

```
func (r * Reader) UnreadByte() error
```

例如：

```
func main() {
    b := []byte("Golang")
    var buf [6]byte
    r := bytes.NewReader(b)           //Buffer 还没被读取过, 返回一个错误.
    err := r.UnreadByte()
    fmt.Println(err)
    n, _ := r.Read(buf[:])           //第一次读取
    fmt.Println(string(buf[:n]), n)
    err = r.UnreadByte()             //取消第一次读取的最后一个字节'g'
    fmt.Println(err)
    n, _ = r.Read(buf[:])            //第二次读取, Buffer 中只剩'g'了.
    fmt.Println(string(buf[:n]), n)
}
```

该例测试结果为：

```
bytes.Reader: at beginning of slice
Golang 6
<nil>
g 1
```

(8) `UnreadRune()` 方法。`UnreadRune()` 方法的作用是取消上次 `ReadRune` 读取的 Unicode 字符；如果上次读取操作不是 `ReadRune`, 则返回错误。该方法原型定义如下：

```
func (r * Reader) UnreadRune() error
```

`UnreadByte` 和 `UnreadRune` 方法的作用是忽略刚才的读操作, 将内部 `offset` 恢复到读操作以前。

例如：

```
func main() {
    b := []byte("中华人民共和国")
    var buf [3]byte
    r := bytes.NewReader(b)
    r.Read(buf[:])                   //用 Read 读取第一个 Unicode 字符'中'
    err := r.UnreadRune()
    fmt.Println(err)
    ch, _, _ := r.ReadRune()         //用 ReadRune 读取第二个 Unicode 字符'华'
    fmt.Println(string(ch))
    err = r.UnreadRune()            //取消上次读取的 Unicode 字符
    fmt.Println(err)
    ch, _, _ = r.ReadRune()         //用 ReadRune 重新读取第二个 Unicode 字符'华'
    fmt.Println(string(ch))
}
```

该例测试结果为：

```
bytes.Reader: previous operation was not ReadRune
华
<nil>
华
```

6.6 程序举例

前面介绍了数字、切片和字典的基本知识，这里列举几个应用事例，进一步加深对这三种数据类型应用的理解。

6.6.1 数组应用

例 6-14 求解约瑟夫环问题，设有 $n(n < 100)$ 个人，其编号分别为 $0, 1, 2, \dots, n$ ，按编号顺序顺时针方向围坐一圈，每人手持一个正整数密码。开始任选一个正整数 m ，从第一个人开始按顺时针报数，报到 m 时停止，报 m 的人出列，并将其手上的密码作为新的 m 值，从他的下一个人开始从 1 重新报数。以此类推，直到所有的人全部出列为止，求出列顺序。

分析题意，该问题可按如下流程处理：

- (1) 输入初始值 m 。
- (2) 使用数组 `psw` 存放密码，下标 i 代表人的编号，`psw[i]` 存放第 i 个人的密码。
- (3) 使用变量 `num` 记录报数，当 $\text{num} = m$ ，则第 i 个人出列，将 `num` 清零重新报数。
- (4) 第 i 个人出列后，将密码 `psw[i]` 赋给 m ，然后将 `psw[i]` 清零。
- (5) 使用变量 k 记录出列人数，当所有人都出列 ($k = n$)，报数终止。

例 6-14 程序代码如下：

```
1 //约瑟夫环问题
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var psw [10]int
10    var num, m, n, k int
11    fmt.Println("请输入总人数 n 和初始值 m:")
12    fmt.Scan(&n, &m)
13    fmt.Println("请输入密码初始化数组 ...")
14    for i := 0; i < n; i++{
15        fmt.Scan(&psw[i])
16    }
17    fmt.Println("密码数组初始化完毕, 开始报数 ...")
```



```

18 Label1:
19     for {
20         for i := 0; i < n; i++ {
21             if psw[i] != 0 {
22                 num++
23                 if num == m {
24                     m = psw[i]
25                     psw[i] = 0
26                     num = 0
27                     fmt.Printf(" %6d", i)
28                     k++
29                     if k == n {
30                         break Label1
31                     }
32                 }
33             }
34         }
35     }
36 }

```

编译并运行该程序,测试过程如下:

请输入总人数 n 和初始值 m:

10 4 ✓

请输入密码初始化数组 ...

4 3 5 2 4 3 5 2 3 4 ✓

密码数组初始化完毕,开始报数 ...

3 5 8 1 6 4 2 9 7 0

6.6.2 Slice 应用

例 6-15 统计文件 golang 中共有多少个单词,假设文件 golang 已经创建好了,内容为“Hello,world! Golang is a beautiful language. I like it!”。

该问题的解决思路及步骤如下:

- (1) 定义字节切片 buf。
- (2) 使用 os 包中的 File 对象的 Read 方法,将文件 golang 的内容读入到字节切片 buf 中。
- (3) 使用 bytes 包中的 FieldsFunc() 函数对文件内容进行断句,比如将标点符号“.”、“,”或“!”作为断句的标识符。
- (4) 对每一个断句再使用 bytes 包中的 Fields() 函数进行分割,这样独立的单词就被分离出来了,每分离出一个单词,单词计数 num 加“1”。
- (5) 输出最终计数结果。

例 6-15 程序代码如下:

```

1 //统计文件中单词的数量
2 package main

```

```

3
4 import(
5     "bytes"
6     "fmt"
7     "os"
8 )
9
10 func main() {
11     var num int
12     buf := make([]byte, 1024)
13     f, _ := os.Open("golang")
14     defer f.Close()
15     n, _ := f.Read(buf)
16     fmt.Println(string(buf[:n]), n)
17     for _, sentence := range bytes.FieldsFunc(buf[:n], f1) {
18         for _, words := range bytes.Fields(sentence) {
19             num++
20             fmt.Printf("%q", words)
21         }
22     }
23     fmt.Println()
24     fmt.Println("文件", f.Name(), "里总共有", num, "个单词.")
25 }
26 func f1(a rune) bool {
27     if a == ',' || a == '.' || a == '!' {
28         return true
29     } else {
30         return false
31     }
32 }

```

编译并运行该程序,输出结果为:

```

Hello,world!Golang is a beautiful language.I like it! 55
"Hello""world""Golang""is""a""beautiful""language""I""like""it"
文件 golang 里总共有 10 个单词.

```

6.6.3 Map 应用

例 6-16 统计今年共多少天,如果是平年,2月28天,全年共365天;如果是闰年,2月29天,全年共366天。

该问题的解决思路及步骤如下:

(1) 定义一个月份天数字典 monthdays, string 类型的月份名作为键, 天数作为值。

(2) 确定今年是平年还是闰年,如果是闰年,将2月份的天数修改为29天。

(3) 使用 for range 语句遍历字典 monthdays, 统计全年总天数。

(4) 输出最终统计结果。

例 6-16 程序代码如下：

```
1 //统计文件中单词的数量
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     monthdays := map[string]int{
10         "Jan": 31, "Feb": 28, "Mar": 31,
11         "Apr": 30, "May": 31, "Jun": 30,
12         "Jul": 31, "Aug": 31, "Sep": 30,
13         "Oct": 31, "Nov": 30, "Dec": 31,
14     }
15     var year string
16     fmt.Println("请输入今年是闰年还是平年 ...")
17     fmt.Scanf("% s", &year)
18     if year == "leap" {
19         monthdays["Feb"] = 29
20     }
21     totleDays := 0
22     for _, days := range monthdays {
23         totleDays += days
24     }
25     fmt.Printf("今年共 %d 天\n", totleDays)
26 }
```

编译并运行该程序，测试过程如下：

请输入今年是闰年还是平年 ...

如果输入：leap ✓

输出结果：今年共 366 天。

如果输入：normal ✓

输出结果：今年共 365 天。

小结

本章主要介绍了 Go 语言的三种引用数据类型：数组、切片和字典，尤其是切片在 Go 程序中应用非常广泛。另外，还介绍了这三种数据类型的定义、创建和初始化。最后通过详尽的事例，介绍了对字节切片进行处理的标准库。

通过这一章的学习，首先需要理解两个基本概念：值类型和引用类型，只有这样才能真正掌握数组、切片和字典的使用方法。然后要掌握切片和字典的基本操作方法，比如添加、查找、删除等。要清楚 new() 和 make() 函数的不同功能，熟练掌握它们。最后要灵活运用 bytes 包中的标准库函数，使用字节切片设计高质量的 Go 应用程序。

习题

6.1 以下关于数组的描述正确的是_____。

- A. 数组的大小是固定的,但可以有不同类型的数组元素
- B. 数组的大小是可变的,但所有数组元素的类型必须相同
- C. 数组的大小是固定的,但所有数组元素的类型必须相同
- D. 数组的大小是可变的,但可以有不同类型的数组元素

6.2 定义数组 `var a [10]int`,对 `a` 的元素引用正确的是_____。

- A. `a[10]`
- B. `a[6.3]`
- C. `a(6)`
- D. `a[10-10]`

6.3 以下不能正确定义数组和赋初值的语句是_____。

- A. `var a=[5]int{1,2,3,4,5}`
- B. `var b=[...]int{1,2,3,4,5}`
- C. `var c [5]int={1,2,3,4,5}`
- D. `var d=[5]int{2: 4,4: 6}`

6.4 阅读下面的程序(假设字符'A'的 ASCII 码值是 65):

```
func main() {  
    var ch = [6]byte{'A', 'B', 'C', 'D', 'E', 'F'}  
    slice1 := ch[:6]  
    slice2 := ch[3:]  
    slice3 := ch[2:4]  
    fmt.Println(slice1)  
    fmt.Println(slice2)  
    fmt.Println(slice3)  
}
```

执行该程序段后, `slice1=_____`,

`slice2=_____`,

`slice3=_____`。

6.5 设字典 `map1` 的键是 `string` 型,值是 `int` 型,初始化为 `[psw1:123 psw2:456 psw3:789]`,要求删除键值对“`psw2:456`”,增加键值对“`psw4:135`”,请编写程序实现上述功能。

6.6 创建一个 `float64` 类型的 `Slice`,初始化并计算该 `Slice` 元素平均值。

6.7 利用 `bytes` 包中的字节切片函数,统计文本文件中含有“`er`”单词的个数,文件读写和单词统计可参考例 6-15。

第7章

函数

函数(Function)是能完成程序预定义功能的代码逻辑单位,一个Go程序至少由一个或多个函数组成。对于可执行的Go程序必须有main()函数,而且只能有一个。

一般来说,Go程序在编译时,函数在程序中所处的位置并不影响编译结果。但为了代码的可读性,一般是从main()函数开始,按函数间的逻辑结构编写代码。编写函数的目的是为了将冗长的代码划分成较小的功能模块,另外一个目的是为了能反复调用程序的某些功能,提高程序代码的复用性。

Go函数不支持嵌套,不支持重载,也不支持默认参数。但Go函数支持变参、多返回值,还可以命名返回值参数。另外,Go语言还支持匿名函数和闭包。在C语言中,如果要在函数定义之前使用它,首先要声明一个函数原型,而在Go语言中无须这么做,Go函数不需要声明函数原型就可以直接使用。

7.1 函数声明

在Go语言中,函数在声明之后就可以使用,无须声明函数原型。Go函数一般由关键字func、函数名、参数列表、返回值、函数体和返回语句组成。当然,有些函数没有参数或返回值。

7.1.1 函数声明基本格式

Go语言也属于静态语言,如变量必须先声明后使用一样,函数在被调用前必须先要声明,否则Go编译器将无法识别。函数声明(Function-declaration)由函数返回类型、函数名和参数列表组成。函数声明的基本格式如下:

```
func functionName (参数列表) 返回值{  
    functionBody  
    :  
    return 语句  
}
```

在声明函数时要注意:

- (1) 函数名的命名规则和变量名相同,遵循标识符命名规则。
- (2) 函数可以有参数或者没有参数,主调用函数通常使用参数向被调用函数传递数据。

(3) 函数体内的所有语句使用一对“{}”括起来,左大括号“{”必须和“func”放在同一行,右大括号“}”必须单独占一行。

(4) 函数会在执行完最后一条语句;或执行 return 语句后结束,Go 语言函数支持多返回值。另外,在终止一个无限循环或 goroutine 时通常也使用 return 语句。

例如,下面的左大括号“{”独占一行声明形式是错误的:

```
func f1()  
{  
}
```

必须写成下面的形式:

```
func f1() {  
}
```

7.1.2 函数声明举例

根据函数的参数和返回值的不同,现举例说明几种常见的函数声明。

例如,无参数且无返回值的函数声明:

```
func f1() {  
    fmt.Println("Hello World!")  
}
```

该例中函数 f1() 的作用仅是输出字符串“Hello World!”,调用时不需要参数,也没有返回值。

例如,有参数但无返回值的函数声明:

```
func f2(a int,b int,c string) {  
    fmt.Println(a,b,c)  
}
```

该例中函数 f2() 在调用时,主调函数要向其传递三个参数,前两个参数为整型,第三个参数是字符串,函数调用成功后输出接收到的数据,但无返回值。在函数声明时,如果多个参数类型一致,可以采用复合定义的形式。即该例中“f2(a int,b int,c string)”,也可以写成“f2(a,b int,c string)”的形式。

例如,有多个参数且有一个返回值的函数声明如下。

```
func f3(a,b int) int {  
    return a + b  
}
```

该例中函数 f3() 在调用时,主调函数要向其传递两个整型参数,函数调用成功后,f3() 向主调函数返回整型值“a+b”。

例如,有多个返回值的函数声明如下。

```
func f4(a,b int) (ret float32,err error) {  
    if b==0 {  
        err = errors.New("Overflow!")  
        return  
    } else {  
        return float32(a) /float32(b),nil  
    }  
}
```

该例中函数 `f3()` 在调用时,主调函数要向其传递两个整型参数 `a` 和 `b`。函数调用成功后有两个返回值,第一个返回值 `ret` 是浮点型,是 `a` 除以 `b` 的结果。第二个返回值 `err` 是错误类型,如果 `b` 等于 0,会返回一个自定义错误“Overflow!”,表示除数不能为 0。否则 `err` 返回 `nil`。

7.2 函数调用

函数调用(Function-call),就是在程序运行过程中使用某个函数完成相关功能的过程。在 Go 语言中,程序是以包为基本单位来组织的(见 1.5.1 节),函数都从属于某个包。所以函数在调用时依据它所属的包不同,分为以下几种情况:

(1) 调用内置函数。Go 语言提供了 13 个内置函数,函数名均为小写,这些函数都可以直接调用,无须导入任何外部包。

(2) 调用标准包中的函数。

(3) 调用一个自定义函数,且与主调函数在同一个包中。

(4) 被调用的函数由用户创建的包提供。

第 2 种和第 4 种情况函数都来自于外部包,而且函数名第一个字母都为大写,所不同的只是标准包由 Go 提供,而用户自己的包要自己创建。在这种情况下,在调用一个函数前都要首先导入外部包,然后才能正常调用,调用的格式为:

```
import "packageName"  
packageName.functionName(参数)
```

第 3 种情况由于被调函数和主调函数处于同一个包中,所以直接调用即可,无须导入外部包,而且函数名首写字母一般是小写,调用格式为:

```
functionName(参数)
```

在上述两种函数调用格式中,参数并非必需的。有些函数在声明时没有参数,所以调用时也就不需要参数了;有些函数声明时定义有参数,那么调用时必须传入参数;有些函数甚至声明为变参形式,调用形式就更灵活了。所以,在进行函数调用时要根据实际情况灵活处理。

7.2.1 调用标准函数

Go 提供了大量的包和实用函数供用户使用,这些函数被称为标准函数。常用的标准包有 `fmt`、`math`、`os`、`time`、`bytes` 等,标准包的信息可以在 Go 安装目录的 `pkg` 下查看,或使用 `godoc` 查看。在调用标准函数时首先要导入该函数所在的包,比如调用 `Println()` 函数要导入 `fmt` 包。

例 7-1 调用时间函数获取当前日期。

```
1 //调用时间函数获取当前日期
2 package main
3
4 import(
5     "fmt"
6     "time"
7 )
8
9 func main() {
10     //获取时间戳
11     t := time.Now()
12     //按年 月 日 时 分秒格式输出
13     fmt.Println(t.String())
14     //按年 月 日格式输出
15     fmt.Println(t.Format("2006 年 01 月 02 日"))
16     //输出星期几
17     fmt.Println(t.Weekday().String())
18 }
```

编译并运行该程序,输出结果为:

```
2013-05-19 18:43:31.59375 + 0800 + 0800
2013 年 05 月 19 日
Sunday
```

本例使用 `time` 包中的 `Now()` 函数获取系统日期,程序的第 6 行导入 `time` 包,程序的第 11 行调用标准函数 `time.Now()`,并返回时间对象 `t`,再使用 `t` 的方法 `String()`、`Format()` 和 `Weekday()` 分别输出日期的“年月日时分秒”格式、“年月日”格式和星期几。

7.2.2 调用自定义函数

通常一个可执行的 Go 程序,首先要构建一个 `main` 包,在 `main` 包中必须声明一个 `main` 函数,然后再声明一些其他自定义函数让 `main` 函数调用,在这种情况下,除非是必需的标准包,否则不用导入任何其他包,直接调用自定义函数就行了。

例 7-2 调用自定义函数。

```
1 //自定义函数的声明与调用
2 package main
```

```
3
4 import(
5     "fmt"
6     "errors"
7 )
8
9 //函数 f1()无参数且无返回值
10 func f1() {
11     fmt.Println("Hello World!")
12 }
13 //函数 f2()有参数但无返回值
14 func f2(a,b int,c string) {
15     fmt.Println(a,b,c)
16 }
17 //函数 f3()有一个返回值
18 func f3(a,b int) int {
19     return a + b
20 }
21 //函数 f4()有两个返回值
22 func f4(a,b int) (ret float32,err error) {
23     if b==0 {
24         err = errors.New("Overflow!")
25         return
26     } else {
27         return float32(a) /float32(b),nil
28     }
29 }
30 func main() {
31     var a,b int = 2,3
32     var c string = "Golang"
33     f1()
34     f2(a,b,c)
35     sum := f3(a,b)
36     fmt.Println(sum)
37     f,err := f4(a,b)
38     fmt.Println(f,err)
39 }
```

编译并运行该程序,输出结果为:

```
Hello World!
2 3 Golang
5
0.6666667 <nil>
```

本例中共定义了4个自定义函数,函数 f1()无参数且无返回值,函数 f2()有参数但无返回值,函数 f3()有一个返回值,函数 f4()有两个返回值。导入 fmt 包是因为要调用 Println()函数,导入 errors 包是因为要调用 New()方法。

7.2.3 调用外部包中的函数

有时在一个 Go 项目中,除了 main 包外还可能创建其他包,如果被调函数是由其他包提供的,此时需导入这个包才能调用相关函数。

在下面的例 7-2 中,首先构建了一个 mymath 包,它提供了 Add()、Sub()、Mult()、Div() 4 个函数,遵循 Go 语言的可见性原则(见 1.5.3 节)这些函数首写字母必须大写。在 main 包中要调用这些函数,只需在调用前导入 mymath 包即可。

构建 mymath 包步骤如下:

- (1) 建立 mymath.go 源文件。
- (2) 编辑 mymath.go 文件,添加源代码,如下所示。

```
1 /*
2 mymath 包实现加减乘除四则运算.
3 加法运算: Add() 函数
4 减法运算: Sub() 函数
5 乘法运算: Mult() 函数
6 除法运算: Div() 函数
7 */
8 package mymath
9
10 func Add(a,b int) int {
11     return a + b
12 }
13
14 func Sub(a,b int) int {
15     return a - b
16 }
17
18 func Mult(a,b int) int {
19     return a * b
20 }
21
22 func Div(a,b int) float32 {
23     if b != 0 {
24         return float32(a) / float32(b)
25     } else {
26         return 0
27     }
28 }
```

- (3) 执行如下命令:

```
go install mymath
```

如果 GOPATH 变量配置正确,则会在 GOPATH 的 pkg 目录下生成 mymath.a 包文件,要导入 mymath 包执行如下命令即可。

```
import "mymath"
```

例 7-3 调用 mymath 包中的函数。

```
1 //调用 mymath 包中的函数
2 package main
3
4 import(
5     "fmt"
6     //导入 mymath 包
7     "mymath"
8 )
9
10 func main() {
11     var a,b int = 2,3
12     add := mymath.Add(a,b)
13     fmt.Println(add)
14     sub := mymath.Sub(a,b)
15     fmt.Println(sub)
16     mult := mymath.Mult(a,b)
17     fmt.Println(mult)
18     div := mymath.Div(a,b)
19     fmt.Println(div)
20 }
```

编译并运行该程序,输出结果为:

```
5
-1
6
0.6666667
```

本例中在程序第 7 行导入 mymath 包,在第 12 行调用 Add() 函数,在第 14 行调用 Sub() 函数,在第 16 行调用 Mult() 函数,在第 18 行调用 Div() 函数。

7.2.4 调用内置函数

除了前面几种函数的调用,Go 语言还提供了一些非常有用的内置函数(Built-in Function),这些函数在调用时无须导入任何包就可以直接使用。内置函数一般都能对不同的数据类型进行操作,比如 len() 函数能获取数组、字符串、切片的长度。有些内置函数直接作用于系统底层,比如像 panic() 函数,通常用于系统错误处理。Go 语言的内置函数虽然不多,但都非常有用,可参见附录 B 的内置函数说明。

7.3 参数传递和返回值

在 Go 语言中,函数参数可以是值类型或引用类型,值类型作为函数参数进行传递时,是一个参数值的拷贝,引用类型作为函数参数传递时,是一个地址拷贝。Go 语言中的函数

被调用时,允许有返回值,甚至是多个返回值。Go 语言还允许定义返回值变量,这样在使用 return 语句进行返回时语句更加简洁。

7.3.1 参数传递

参数传递(Parameter-passing),是指在程序运行过程中,实际参数就会将参数值传递给相应的形式参数,然后在函数中实现对数据处理和返回的过程。最常见的参数传递方法有按值传递参数,按地址传递参数或按数组传递参数。

在调用函数时,大多数情况下,主调函数和被调函数之间都有数据传递关系。在声明函数时函数名后面括号内中的变量名称为形式参数(Formal Parameter),简称形参。在主调函数中调用另一个函数时,函数名后面括号中的参数称为实际参数(Actual Parameter),简称实参。

说明:

(1) 在声明函数时指定的形参,在未进行函数调用时并不占用存储单元。在进行函数调用时,形参才被分配存储单元,在调用结束后存储单元被释放。

(2) 实参可以是常量、变量或者表达式,不管是哪一种形式都必须有确定的值,在调用时将实参的值赋给形参。

(3) 在声明函数时,必须指定形参的类型,而且实参与形参的类型必须一致。

1. 常规传递

当使用普通变量作为函数参数时,在传递参数时只是对变量值的拷贝,即将实参的值复制给变参。当函数对变参进行处理时,并不影响原来实参的值。

例如:

```
func main() {  
    var b int = 1  
    f1(b)  
    fmt.Println(b)  
}  
  
func f1(a int) {  
    a = 2  
    fmt.Println(a)  
}
```

该例中 int 型变量 a 是函数 f1() 的形参,在 main() 函数中调用 f1() 时, int 型变量 b 作为实参传递给函数 f1()。函数 f1() 在接收到实参 b 后,只是将 b 的值复制给形参 a,所以函数 f1() 对 a 的任何操作都不会影响 b 原来的值,程序的最后运行结果为: 2,1。

2. 指针传递

函数的参数不仅可以使普通变量,还可以使用指针变量。当使用指针变量作为函数的参数时,在进行参数传递时将是一个地址拷贝,即将实参的内存地址复制给变参,这时对变参的修改同时也将会影响到实参的值。

例如上例代码修改如下：

```
func main() {
    var b int = 1
    f2(&b)
    fmt.Println(b)
}

func f2(a *int) {
    *a = 2
    fmt.Printf("%d\n", *a)
}
```

该例中 `int` 型指针变量 `a` 作为函数 `f2()` 的形参，在 `main()` 函数中调用 `f2()` 时，使用取地址符“&”获取变量 `b` 的内存地址作函数 `f2()` 的实参，实参 `b` 的内存地址会复制给形参 `a`，当对 `a` 进行操作时，其实质就是直接在内存中对 `b` 进行操作，程序最后运行结果为：2、2。

3. 数组元素作为函数参数

前面已经介绍了可以使用普通变量作为函数的参数，显然，数组元素也可以作为函数的参数，因为数组就是同一种变量的集合。当使用数组元素作为函数参数时，其使用方法和普通变量相同，即是一个“值拷贝”。

例如：

```
func main() {
    var b = [5]int{1,2,3,4,5}
    f3(b[2])
    fmt.Println(b[2])
}

func f3(a int) {
    a += 1
    fmt.Println(a)
}
```

该例中将数组元素 `b[2]` 作为函数 `f3()` 的实参，即将 `b[2]` 的值复制给形参 `a`，对形参 `a` 的操作不会影响数组元素 `b[2]` 的值，程序运行结果为：4、3。

4. 数组名作为函数参数

除了数组元素可以作为函数参数外，数组名也可以作为函数的参数。和其他语言不同的是，Go 语言在将数组名作为函数参数时，参数传递即是对数组的复制。在形参中对数组元素的修改，都不会影响实参数组元素原来的值。

例如：

```
func main() {
    var b = [5]int{1,2,3,4,5}
    f4(b)
    fmt.Println(b[0])
}
```

```
}  
func f4(a [5]int) {  
    a[0] += 1  
    fmt.Println(a[0])  
}
```

该例中直接使用数组 b 的名字作为函数 f4() 的参数,在形参中对 a[0] 的修改不会影响实参 b[0] 中的值,程序运行结果为: 2、1。

而在其他语言中,比如 C 语言,数组名作为实参传递的是数组首元素的地址,对 a[0] 的修改将会影响到 b[0] 的值。所以,在 Go 语言中使用数组名作为函数参数时,要注意以下几点:

(1) 在使用数组名作为函数参数时,应在主调函数和被调函数中分别定义数组,不能只在一方定义。

(2) 在使用数组名作为函数参数时,实参数组和形参数组类型必须一致。比如实参 b 的类型是 [5]int,形参 a 也应是 [5]int。形参 a 定义成 []int、[10]int 等类型都是错误的,而 C 语言往往允许这么做。Go 语言是类型安全的,[5]int 和 []int、[10]int 永远是不同的类型。

(3) 在使用数组名作为函数参数时,其传递过程是对数组的“值拷贝”,即将实参数组完全复制给形参数组。

5. Slice 作为函数参数

当希望通过形参对底层数组进行修改时,可以使用 Slice 作为函数参数。在使用 Slice 作为函数参数时,进行参数传递将是一个地址拷贝,即将底层数组的内存地址复制给参数 Slice。这时,对 Slice 元素的操作即是对底层数组元素的操作。

例如:

```
func main() {  
    var b = [5]int{1,2,3,4,5}  
    var s1 []int = b[:]  
    f5(s1)  
    fmt.Println(b[0])  
}  
func f5(s []int) {  
    s[0] += 1  
    fmt.Println(s[0])  
}
```

该例中使用 Slice 作为函数 f5() 的形参,在进行参数传递时,Slice 复制的是底层数组的内存地址。所以对形参中 Slice 元素的修改,同时也会修改底层数组元素的值,程序运行结果为: 2、2。

6. 函数作为参数

在 Go 语言中,函数也作为一种数据类型,所以函数也可以作为函数的参数来使用。

例如：

```
func main() {
    var a,b int = 3,4
    f := sum
    f6(a,b,f)
}
func f6(a,b int,sum func(int,int) int) {
    fmt.Println(sum(a,b))
}
func sum(a,b int) int {
    return a + b
}
```

该例中函数 `sum` 作为函数 `f6()` 的形参，而变量 `f` 是一个函数类型，作为 `f6()` 调用时的实参，程序运行结果为：7。

7.3.2 返回值

通常，主调函数在调用被调函数时都希望能得到一个确定的值，这个值就是函数的返回值(Return-value)。带返回值的函数在声明的同时要定义返回值的类型，返回值的类型可以是 Go 语言支持的基本数据类型。

1. return 语句

Go 语言中函数的返回值是通过 `return` 语句获得的。`return` 语句将被调函数得到的一个确定的值带回主调函数。

例如：

```
func main() {
    sum := f1(3,6)
    fmt.Println(sum)
}
func f1(a,b int) int {
    return a + b
}
```

该例中函数 `f1()` 调用成功后，`return` 语句返回一个 `int` 型的结果 9。

2. 多返回值

Go 语言支持函数可以有多个返回值，这和 C 语言有明显不同，多返回值使得 Go 程序设计起来更加灵活，而且功能强大。如果函数定义了多个返回值，除了说明各个返回值的类型以外，还需使用一对括号将它们括起来。

例如：

```
func main() {
    sum,div := f2(3,6)
```



```
    fmt.Println(sum, div)
}
func f2(a, b int)(int, float32) {
    return a * b, float32(a)/float32(b)
}
```

该例中函数 `f2()` 调用成功后, `return` 语句返回两个结果, 一个是 `a * b` 的值 9, 为 `int` 型; 一个是 `a/b` 的值 0.5, 为 `float32` 型。

3. 返回值的忽略

在处理多返回值时, 对于不想要的值, 可以使用空白标识符 (Blank-identifier) “`_`” 进行忽略。比如上例中, 主调函数在调用函数 `f2()` 时如果只需要结果 `a * b`, 则可以使用标识符 “`_`” 忽略掉结果 `a/b`。

例如:

```
func main() {
    rst, _ := f2(3, 6)
    fmt.Println(rst)
}
func f2(a, b int)(int, float32) {
    return a * b, float32(a)/float32(b)
}
```

该例中 `main()` 函数在调用 `f2()` 时使用 “`_`” 忽略了函数的第二个返回值, 所以输出结果为 18。

4. 命名返回值参数

Go 语言还支持命名返回值参数, 如果使用命名返回值参数, 则 `return` 语句可以为空。否则, `return` 语句必须按照顺序返回多个结果。

例如:

```
func main() {
    sum, sub := f3(3, 6)
    fmt.Println(sum, sub)
}
func f3(a, b int) (sum, sub int) {
    sum = a + b
    sub = a - b
    return
}
```

该例中函数 `f3()` 有两个返回值参数, 一个是 `sum`, 一个是 `sub`。命名了返回值参数, 函数 `f3()` 使用一条空 `return` 语句, 就可以向主调函数返回 `sum`、`sub` 这两个结果。如果未命名返回值参数, 函数 `f3()` 的返回语句写法是: `return a+b, a-b`。

7.4 变参函数

在使用 Go 语言编程时,函数中形式参数的数目通常是确定的,在调用时要依次传递与形式参数对应的所有实际参数。但在某些情况下希望函数的参数个数可以根据实际需要来确定,这就是变参函数(Variable-argument Function)。最典型的变参函数有 `fmt.Printf()`、`fmt.Scanf()` 和系统调用 `exec.Command()` 等。

7.4.1 变参函数的声明

Go 语言支持不定长变参,但是要注意不定长变参只能作为函数的最后一个参数,不能放在其他参数的前面。变参函数的声明格式如下:

```
func functionName (variableArgumentName ... dataType) 返回值{  
    functionBody  
    :  
}
```

说明:

(1) 变参类型的定义格式是“... 类型”,而且变参必须是函数的最后一个参数。如果函数还有其他参数,则必须放在变参的前面定义,例如 `func f1(a int,s string,args...int){}`。

(2) 不定长变参其实质就是一个切片,可以使用 `range` 进行遍历。

例如:

```
func f1(args ... int) {  
    for _,arg := range args {  
        fmt.Println(arg)  
    }  
}
```

上例中函数 `f1()` 是变参函数, `args` 是函数 `f1()` 的整型不定长变参, `args` 可以使用 `range` 进行遍历。函数 `f1()` 的调用方式如下:

```
f1(1,2,3)  
f1(1,2,3,4,5)
```

例 7-4 变参函数的声明和调用。

```
1 //变参函数的声明  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7
```

```
8 func main() {
9     f1(1,2,3)
10    f1(1,2,3,4,5)
11 }
12 /*
13  声明函数 f1() 为变参函数
14 */
15 func f1(args ...int) {
16     //变参 args 其实是一个切片
17     fmt.Println(args)
18 }
```

编译并运行该程序,输出结果为:

```
[1 2 3]
[1 2 3 4 5]
```

例 7-4 中声明函数 `f1()` 为变参函数,然后输出变参 `args` 的值,证明变参 `args` 其实就是一个切片,即 `args...int` 变参形式的定义,等价于 `args []int` 切片形式的定义。

7.4.2 变参的传递

不定长变参在进行参数传递时,接收到的是某一种类型的 Slice,所以不定长变参在进行传递时可以全部传递,也可以部分传递。如例 7-5 演示了变参的两种传递方式。

例 7-5 变参的传递。

```
1 //变参的传递
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     f1(1,2,3,4,5)
10 }
11 /*
12  声明函数 f1()、f2() 为变参函数
13 */
14 func f1(args...int) {
15     //全部传递,即将函数 f1() 的 args 直接复制给函数 f2().
16     f2(args...)
17     //部分传递,将 args 中的第 3 个以后的元素复制给函数 f2().
18     f2(args[2:]...)
19 }
20 func f2(args...int) {
21     fmt.Println(args)
22 }
```


编译并运行该程序,输出结果为:

```
[1 2 3 4 5]
[3 4 5]
```

例 7-5 中声明函数 `f1()`、`f2()` 均为变参函数,函数 `f1()` 调用 `f2()` 并将自己的变参传递给 `f2()`。比较例 7-4 和例 7-5 发现,使用变参传递可以使变参函数的应用更加灵活。

不定长变参在进行参数传递时虽然接收到的是一个 Slice,但是和直接传递一个 Slice 还是有所区别的。直接传递一个 Slice 是一个“地址拷贝”(见 7.3.1 节),而不定长变参仍然是一个“值拷贝”。即不定长变参在传递一个 Slice 时,它仅仅是获取 Slice 的一个副本,而当对这个 Slice 副本进行操作时,肯定不会改变原来 Slice 的值。

7.4.3 任意类型的变参

前面的例子中变参中的元素都必须是同一种类型,比如同为整型、浮点型、字符串等,但在实际应用中,用户希望向变参函数传递不同类型的参数,比如向 `fmt.Printf()` 函数那样。Go 语言规定,如果希望传递任意类型的变参,变参类型应指定为空接口 `interface{}`。

例如:

```
func f1(args... interface{}) {
    :
}
```

在 Go 语言中,空接口 `interface{}` 可以指向任何数据对象,所以可以使用 `interface{}` 定义任意类型变参(见第 9 章)。同时,`interface{}` 也是类型安全的。

例 7-6 任意类型变参函数的声明和调用。

```
1 //任意类型变参
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     f1(2, "Go", 8, "Language", 'a', false, 'A', 3.14)
10 }
11 /*
12  声明函数 f1() 为任意类型变参
13 */
14 func f1(args... interface{}) {
15     var num = make([]int, 0, 6)
16     var str = make([]string, 0, 6)
17     var ch = make([]int32, 0, 6)
18     var other = make([]interface{}, 0, 6)
19     for _, arg := range args {
```

```
20     switch v := arg.(type) {
21     case int:
22         num = append(num, v)
23     case string:
24         str = append(str, v)
25     case int32:
26         ch = append(ch, v)
27     default:
28         other = append(other, v)
29     }
30 }
31 fmt.Println(num)
32 fmt.Println(str)
33 fmt.Println(ch)
34 fmt.Println(other)
35 }
```

编译并运行该程序,输出结果为:

```
[2 8]
[Go Language]
[97 65]
[false 3.14]
```

例 7-6 中声明函数 `f1()` 为任意类型变参函数,在函数 `f1()` 中创建了 4 个切片 `num`、`str`、`ch` 和 `other`,这 4 个切片用来统计函数 `f1()` 从主调函数接收的变参。如果是整型变参,就添加到切片 `num` 中;如果是字符串类型,就添加到切片 `str` 中;如果是字符类型,就添加到切片 `ch` 中;如果是其他类型,都添加到切片 `other` 中。最后打印这 4 个切片,查看统计结果。通过输出结果可以看出,布尔型“false”和浮点型“3.14”都被统计到 `other` 中。

7.5 匿名函数与闭包

在 Go 语言中,所有的函数都是值类型,可以作为参数传递。Go 语言支持常规的匿名函数和闭包,可以随意对匿名函数变量进行传递和调用。

7.5.1 匿名函数

匿名函数(Anonymous-function)是指不需要定义函数名的一种函数实现方式。匿名函数最早出现在 LISP 语言中,目前 PHP、JavaScript 都支持这种函数形式。

在 Go 语言中,函数可以像变量一样被传递或使用,这与 C 语言的函数回调比较类似。不同的是,Go 语言支持随时在代码里定义匿名函数。匿名函数由一个不带函数名的函数声明和函数体组成,如下所示:

```
func (参数列表) 返回值{
    functionBody
}
```

```
    :  
    return 语句  
}
```

例如:

```
func(a,b int) int {  
    return a + b  
}
```

上例声明了一个匿名函数,该匿名函数有两个整型参数,返回值也为整型。匿名函数在调用时,可以直接赋值给一个变量,或者直接执行。

例 7-7 匿名函数的声明与调用。

```
1 //匿名函数的声明与调用  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7  
8 func main() {  
9     //声明并将匿名函数赋值给变量 f  
10    f := func(a,b int) int {  
11        return a + b  
12    }  
13    //对函数类型变量 f 进行调用  
14    sum := f(2,3)  
15    fmt.Println(sum)  
16    //声明并直接执行匿名函数  
17    sum = func(a,b int) int {  
18        return a + b  
19    }(5,7)  
20    fmt.Println(sum)  
21 }
```

编译并运行该程序,输出结果为:

```
5  
12
```

例 7-7 演示了匿名函数的声明方法,并通过两种方式对匿名函数进行了调用。第一种将匿名函数赋值给一个变量,也说明了在 Go 语言中函数也被当作一种数据类型来使用。程序的第 19 行,右大括号“}”后直接跟参数列表表示函数调用,说明了匿名函数在声明的同时可以直接执行。

在 Go 语言中,使用匿名函数时要注意它不能作为顶级函数,而只能放在其他函数的函数体中,也就是说它必须有一个外层函数。

7.5.2 闭包

闭包(Closure),就是内部函数通过某种方式使其可见范围超出了其定义的范围,这就产生了一个在其定义范围内的闭包。在 Go 语言中,闭包可以作为函数对象或者匿名函数,也就是说 Go 语言中的闭包同样也会引用到函数外的变量。闭包的实现确保只要闭包还被使用,那么被闭包引用的变量会一直存在。

例如:

```
func main() {  
    f := closures(10)  
    fmt.Println(f(1))  
    fmt.Println(f(2))  
}  
func closures(x int) func(int) int {  
    return func(y int) int {  
        return x + y  
    }  
}
```

这段代码有三个特点:

- (1) 匿名函数嵌套在函数 closures 内部。
- (2) 函数 closures 返回匿名函数。
- (3) 匿名函数引用了自身外部的变量 x。

函数 closures 内的匿名函数就是上面所说的内部函数,这样在执行完 `f := closures(10)` 后,变量 f 实际上是指向了匿名函数,所以执行 `f(1)` 后输出 11,执行 `f(2)` 后输出 12。这段代码其实就创建了一个闭包,因为函数 closures 的变量 f 引用了函数 closures 内的匿名函数。也就是说:当函数 closures 的内部函数(匿名函数)被函数 closures 外的一个变量引用的时候,就创建了一个闭包。

例 7-8 函数的闭包。

```
1 //函数的闭包  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7  
8 func main() {  
9     f := closures(10)  
10    fmt.Println(f(1))  
11    fmt.Println(f(2))  
12 }  
13 //注意函数的闭包返回值是匿名函数  
14 func closures(x int) func(int) int {  
15     return func(y int) int {
```

```
16 //注意内部函数引用的参数超出了它的作用范围
17 return x + y
18 }
19 }
```

编译并运行该程序,输出结果为:

```
11
12
```

通过对函数闭包的原理分析和实例测试,函数闭包主要有以下几个作用:

(1) 函数闭包可以保护函数内的变量安全。比如例 7-8 中函数 closures 的参数 x 只有内部函数才能访问,而无法通过其他途径访问到。

(2) 函数闭包可以在内存中维持一个变量。比如上例中函数 closures 的参数 x 在定义变量 f 时被初始化为 10,然后就一直保持为这个值,后面无论调用多少次 f, x 在内存中一直存在。

7.6 函数的递归调用和 defer 语句

在 Go 语言中,函数的递归调用和 defer 调用语句都属于特殊的用法,本节主要介绍这两种用法的特点和使用场合。

7.6.1 函数的递归调用

一个含有直接或间接地调用自身语句的函数,称为递归函数(Recursibe-function)。递归函数调用自身的过程,称为递归调用(Recursibe-call)。

下面的例子为函数直接调用自身:

```
func f() {
    f()
}
```

下面的例子为函数间接调用自身:

```
func f1() {
    f2()
}
func f2() {
    f1()
}
```

从上面两个例子可以看到,这两种递归调用都是无终止的自身调用。显然,程序中不应出现这种无终止的递归调用,只应出现有限次的、有终止的递归调用。即递归函数必须满足以下两个条件:

(1) 递归函数在每一次调用自己时,必须是(在某种意义上)更接近于最终结果。

(2) 必须有一个终止递归调用的条件控制。

所以,函数的递归调用通常使用 if 语句来控制,只有当某一条件成立时才继续执行递归调用,否则就不再继续。

例 7-9 使用递归函数实现例 6-3 的 Fibonacci 数列。

```
1 //使用递归函数求 Fibonacci 数列
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var result int = 0
10    for i := 0; i < 20; i++{
11        result = fibonacci(i)
12        if i%5 == 0 {
13            fmt.Printf("\n")
14        }
15        fmt.Printf("% 4d ", result)
16    }
17 }
18 //fibonacci()函数递归调用
19 func fibonacci(n int) (res int) {
20     if n <= 1 {
21         res = 1
22     } else {
23         res = fibonacci(n-1) + fibonacci(n-2)
24     }
25     return
26 }
```

编译并运行该程序,输出结果为:

1	1	2	3	5
8	13	21	34	55
89	144	233	377	610
987	1597	2584	4181	6765

7.6.2 defer 语句

在 Go 语言中,可以使用关键字 defer 向函数注册退出调用,即当主调函数退出时,defer 后的函数才会被调用。defer 语句的作用是不管程序是否出现异常,均在函数退出时自动执行相关代码。

例如:


```
func main() {  
    defer fmt.Println("The first.")  
    fmt.Println("The second.")  
}
```

上例首先输出“The second.”,然后才输出“The first.”。

1. defer 语句实现函数逆序调用

如果程序中有多个 defer 语句,则按照“先进后出(FILO)”的次序执行,即最后一个 defer 语句将最先被执行。

例如:

```
func main() {  
    for i := 0; i < 5; i++ {  
        defer fmt.Println(i)  
    }  
}
```

上例的输出结果是:4、3、2、1、0。

2. defer 语句支持匿名函数调用

在 Go 语言中,defer 语句还支持匿名函数调用,如果函数有返回值,被延迟执行的匿名函数还会读取函数的返回值,并对返回值赋值。

例 7-10 defer 语句定义函数延迟执行。

```
1 //defer 语句定义函数延迟执行  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7  
8 func main() {  
9     f1()  
10    f2()  
11    fmt.Printf("\n")  
12    fmt.Println(f3())  
13 }  
14 //一个延迟执行的函数的变量的值在声明延迟时别赋值  
15 func f1() {  
16     i := 0  
17     defer fmt.Println(i)  
18     i++  
19     return  
20 }  
21 //被延迟的函数按照先进后出的顺序执行
```

```
22 func f2() {
23     for i := 0; i < 4; i++{
24         defer fmt.Printf("%d ", i)
25     }
26 }
27 //被延迟的匿名函数会读取函数 f3 的返回值,或对 f3 的返回值赋值.
28 func f3() (i int) {
29     defer func() {
30         i++
31     }()
32     return 1
33 }
```

编译并运行该程序,输出结果为:

```
0
3 2 1 0
2
```

3. defer 语句用于清理工作

在 Go 程序中,当程序返回或发生异常时,defer 语句通常用来做一些函数调用后的清理工作,释放资源变量。

例如关闭文件句柄:

```
srcFile, err := os.Open("myFile")
defer srcFile.Close()
```

关闭互斥锁:

```
mutex.Lock()
defer mutex.Unlock()
```

上面例子中 defer 语句的用法有两个优点:

(1) 让设计者永远也不会忘记关闭文件,有时当函数返回时常常忘记释放打开的资源变量。

(2) 将关闭和打开靠在一起,程序的意图也变得清晰很多。

例 7-11 defer 语句保证文件能够正常关闭。

```
1 //defer 语句保证文件能够正常关闭
2 package main
3
4 import(
5     "fmt"
6     "os"
7     "io"
```

```
8 )
9
10 func main() {
11     copylen, err := copyFile("dst.txt", "src.txt")
12     if err != nil {
13         return
14     } else {
15         fmt.Println(copylen)
16     }
17 //函数 copyFile 的功能是将源文件 src 的数据复制给目标文件 dst
18 func copyFile(dstName, srcName string) (copylen int64, err error) {
19     src, err := os.Open(srcName)
20     if err != nil {
21         return
22     }
23     //当 return 时就会调用 src.Close()把源文件关闭.
24     defer src.Close()
25     dst, err := os.Create(dstName)
26     if err != nil {
27         return
28     }
29     //当 return 时就会调用 src.Close()把目标文件关闭.
30     defer dst.Close()
31     return io.Copy(dst, src)
32 }
```

编译并运行该程序,输出结果为:

25

7.6.3 异常恢复机制

Go 没有 Java 中那种 try-catch-finally 结构化异常处理机制,而是使用 panic()函数代替 throw/raise 引发错误,然后在 defer 语句中调用 recover()函数捕获错误,这就是 Go 语言的异常恢复机制——panic-and-recover 机制。

panic 是一个内置函数,可以中断原有的控制流程,进入一个异常流程中。当函数调用 panic 时,函数的执行将被中断,并且函数中的延迟函数会正常执行,然后函数返回到调用它的地方。在调用的地方,函数的行为就像调用了 panic。这一过程继续向上,直到程序崩溃时的所有 goroutine 返回。异常可以直接调用 panic 产生,也可以由运行时错误产生,例如访问越界的数组。

Recover 也是一个内置的函数,它可以让进入异常流程中的 goroutine 恢复过来。Recover 仅在延迟函数中有效。在正常的执行过程中,调用 recover 会返回 nil 并且没有其他任何效果。如果当前的 goroutine 陷入异常,调用 recover 可以捕获到 panic 的输入值,并且恢复正常的执行。

例 7-12 使用 panic-and-recover 机制处理参数越界问题,比如函数 f() 有一个整型参数,当 f() 被调用时如果传入的参数小于 100,则属于正常范围,函数正常调用退出;否则参数越界,函数调用异常,报错退出。

例 7-12 代码如下:

```
1 //panic - and - recover 异常恢复机制
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     //必须要先声明 defer,否则捕获不到 panic 异常
10    defer func() {
11        fmt.Println("函数 defer 开始运行...")
12        if err := recover(); err != nil {
13            //这里的 err 就是 panic 传入的内容
14            fmt.Println("程序异常退出: ",err)
15        } else {
16            fmt.Println("程序正常退出.")
17        }
18    }()
19    f(101)
20 }
21 func f(a int) {
22    fmt.Println("函数 f 开始运行...")
23    if a > 100 {
24        panic("参数值超出范围!")
25    } else {
26        fmt.Println("函数 f 调用结束.")
27    }
28 }
```

编译并运行该程序,当参数为 10(小于 100)时输出结果为:

```
函数 f 开始运行...
函数 f 调用结束.
函数 defer 开始运行...
程序正常退出.
```

编译并运行该程序,当参数为 101(大于 100)时输出结果为:

```
函数 f 开始运行...
函数 defer 开始运行...
程序异常退出: 参数值超出范围!
```

7.7 程序举例

前面介绍了函数声明、调用、参数传递等基础知识,这里列举几个应用事例,以进一步加深对函数应用的理解。

7.7.1 函数嵌套调用举例

例 7-13 使用函数嵌套调用,求正整数 a 和 b 的最小公倍数。

程序功能分析: 正整数 a 、 b 的最小公倍数和最大公约数之间有关系,设 a 、 b 的最大公约数为 d ,最小公倍数为 c ,则 $c=(a \times b)/d$ 。

所以,本例需要定义两个函数: 求最大公约数函数 `divisor()` 和求最小公倍数函数 `multiple()`,函数 `multiple()` 嵌套调用函数 `divisor()`。程序代码如下:

```
1 //函数嵌套调用求最小公倍数
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     var a,b,c int
10    fmt.Println("请输入正整数 a 和 b: ")
11    fmt.Scanf("%d %d",&a,&b)
12    c = multiple(a,b)
13    fmt.Printf("正整数 %d 和 %d 的最小公倍数为 %d.\n",a,b,c)
14 }
15 /*
16    函数 divisor()的功能是求最大公约数
17 */
18 func divisor(a,b int) int {
19     var r int
20     for {
21         r = a % b
22         if r != 0 {
23             a = b
24             b = r
25         } else {
26             break
27         }
28     }
29     return b
30 }
31 /*
32    函数 multiple()的功能是求最小公倍数
33 */
```



```
34 func multiple(a,b int) int {  
35     var d int  
36     d = divisor(a,b)  
37     return (a * b / d)  
38 }
```

编译并运行该程序,测试过程如下:

请输入正整数 a 和 b:

如果输入: 21,9 ✓

输出结果: 正整数 21 和 9 的最小公倍数为 63.

如果输入: 13,7 ✓

输出结果: 正整数 13 和 7 的最小公倍数为 91.

7.7.2 变参函数举例

例 7-14 使用变参函数求最小数。

程序功能分析: 该问题要解决的是要在一组整型数中找出最小数,但整型数数量未知,所以要将最小数函数 Min()设计成变参形式。Min()函数的参数可以是一组立即数,也可以是整型数组。该程序代码如下:

```
1 //使用变参函数求最小数  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7  
8 func main() {  
9     min := Min(12,3,24,7)  
10    fmt.Println("最小数为: ",min)  
11    arr := []int{-6,13,7,25,-13}  
12    min = Min(arr...)  
13    fmt.Println("数组中的最小数为: ",min)  
14 }  
15 /*  
16     函数 Min()的功能是找出最小数  
17 */  
18 func Min(a ...int) int {  
19     if len(a) == 0 {  
20         return 0  
21     }  
22     min := a[0]  
23     for _, v := range a {  
24         if v < min {  
25             min = v  
26         }  
27     }  
28 }
```



```
27     }
28     return min
29 }
```

编译并运行该程序,输出结果为:

最小数为: 3

数组中的最小数为: -13

7.7.3 多返回值函数举例

例 7-15 使用多返回值函数找出字节切片中的整型数。

程序功能分析: 该问题要解决的是从一个指定位置找出字节切片中的整型数,并返回下个数的索引位置。所以在编写查找函数时,它至少要返回两个值,一个是找到的整型数,另一个是下个数的索引位置。

字节切片中的元素可以看作是 ASCII 字符,如果是字符‘0’~‘9’,就认为它是整型数的某个数位;如果是其他字符,则忽略。在本例中,函数 nextInt()的主要功能就是,将连续挨在一起的字符‘0’~‘9’转换成整型数并返回。遇到非‘0’~‘9’字符,则意味着上一个数的结束,接着查找下一个整数的起始位置。该程序代码如下:

```
1 //使用多返回值函数找出字节切片中的整型数
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     a := []byte{'1','2','3','4','z','x',
10                '5','6','d','g','7','8'}
11     var index, x, start int
12     fmt.Println("请输入起始索引位置...")
13     fmt.Scanf("%d", &start)
14     for {
15         index, x = nextInt(a, start)
16         fmt.Printf("找到整数 %d, 下一个数的位置从 %d 开始.\n", x, index)
17         if index != 0 && index < len(a) {
18             start = index
19             continue
20         } else {
21             break
22         }
23     }
24 }
25 /*
26 函数 nextInt()的功能是从指定位置找出字节切片中的整数,
```

```
27 并返回下一个数的索引位置.
28 */
29 func nextInt(b []byte, i int) (index, x int) {
30     var start bool
31     if b[i] >= '0' && b[i] <= '9' {
32         start = true
33     } else {
34         start = false
35     }
36     for ; i < len(b); i++ {
37         if b[i] >= '0' && b[i] <= '9' {
38             if start == false {
39                 index = i
40                 break
41             } else {
42                 x = x * 10 + int(b[i]) - '0'
43             }
44         } else if b[i] < '0' || b[i] > '9' {
45             start = false
46         }
47     }
48     return index, x
49 }
```

编译并运行该程序,测试过程如下:

请输入起始索引位置...

如果输入: 0 ✓

输出结果: 找到整数 1234, 下一个数的位置从 6 开始.

找到整数 56, 下一个数的位置从 10 开始.

找到整数 78, 下一个数的位置从 0 开始.

如果输入: 2 ✓

输出结果: 找到整数 34, 下一个数的位置从 6 开始.

找到整数 56, 下一个数的位置从 10 开始.

找到整数 78, 下一个数的位置从 0 开始.

小结

本章主要介绍了 Go 语言的函数的定义、声明和调用,另外还介绍了变参函数、匿名函数和闭包。和其他高级程序设计语言相比,Go 语言的函数功能还是比较强大的,这主要体现在 Go 不但实现了其他高级语言中函数的基本功能,Go 还将函数的功能拓展到对象方法,这将在第 8 章重点进行讲述。最后,在 Go 语言中函数还被当作一种数据类型,用户可以定义函数变量,使函数的应用方式更加灵活多样。

通过这一章的学习,首先要掌握函数的基本声明方法和调用方法,要熟练掌握函数的参数传递和多返回值特性,要灵活运用变参函数、匿名函数、闭包和 defer 调用这几种特殊用法。

习题

7.1 已知三角形三边 a 、 b 、 c ，编写函数求三角形面积公式 $\text{area}()$ ，求三角形面积。（三角形面积公式为 $\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$ ， s 为周长一半）

7.2 编写函数 $\text{find}()$ ，在整型数组 arr 中查找最大元素下标 max 和最小元素下标 min ，要求数组作为函数参数，且使用多返回值同时返回查找结果 max 和 min 。

7.3 编写一个函数，用于计算 float64 类型 Slice 的元素平均值，要求将 Slice 作为函数参数传递给处理函数。

7.4 设计一个变参函数 $\text{max}()$ ，该函数可以找出不定数量整型数中的最大值。

7.5 编写函数 $\text{fac}()$ ，使用递归调用计算 $n!$ ， $n!$ 公式为：

$$n! = \begin{cases} 1 & (n=0) \\ n(n-1)! & (n>0) \end{cases}$$

7.6 在 Go 语言中， defer 语句的作用是延迟调用，阅读下述代码：

```
func main() {
    for i := 0; i < 5; i++ {
        defer fmt.Printf("%d ", i)
    }
}
```

上述程序运行后的输出结果是：_____。

7.7 使用 panic-and-recover 机制编写一个检测用户账号合法性的程序，比如用户中不能出现空格，不能出现非法字符，比如‘，’、‘#’、‘!’等，可参考例 7-12。

第8章

结构体和方法

前面的章节已经介绍了 Go 语言的基本数据类型,以及一种构造数据类型——数组,数组中的各元素是属于同一种类型的。有时需要将不同类型的数据项组织在一起,以便于引用,这种数据类型就是结构体(Struct)。结构体中的数据项,通常被称为字段(Field)。

结构体中的字段都是互相联系的,比如一个学生结构体里包含学号、姓名、年龄、性别、班级等字段(如图 8-1 所示),这些字段都是与某个学生相联系的。

学号	姓名	性别	年龄	班级
13001	李明	男	18	网络01

图 8-1 学生信息结构体

8.1 结构体的定义

从图 8-1 可以看出,结构体中各字段的数据类型通常各不相同,这就需要用户在程序中构造自己所需的结构体。

8.1.1 结构体定义

在 Go 语言中,可以使用关键词“type”定义新的数据类型,由于每一个结构体都属于一个新的数据类型,所以在定义结构体时必须使用 type 关键字,另外还有关键字“struct”。

结构体的基本定义格式如下:

```
type structName struct{
    field1 类型
    field2 类型
    :
}
```

在定义结构体时要注意:

- (1) 结构体的命名规则和变量名相同,遵循标识符命名规则。
- (2) 结构体的命名规则还遵循可见性命名规则,即只有首写字母为大写的结构体和属性才能在包外被访问。

(3) 结构体内的所有属性使用一对“{}”括起来,左大括号“{”必须和关键字 struct 放在同一行,右大括号“}”必须单独占一行。

1. 基本类型字段

在定义结构体时,字段的数据类型可以是 Go 语言支持的任何基本数据类型,比如: int 型、float 型、string 型等。例如:

```
type student struct{
    id int
    name string
    sex bool
    age int
    class string
}
```

该例定义了一个学生结构体 student,有 5 个字段:学号 id 为 int 型,姓名 name 为 string 型,性别 sex 为 bool 型,年龄 age 为 int 型,班级 class 为 string 型。

2. 预留字段

在定义结构体时,每个字段都应有一个名字。如果某些字段根本就不会被使用,只是为了以后对结构体进行扩展预留的,这种字段叫做预留字段(Reserved-field)。在 Go 语言中,预留字段使用标识符“_”来命名。例如:

```
type people struct {
    name string
    age int
    _ string
}
```

该例定义了一个 people 结构体,其中第三个字段是预留字段。

3. 结构体作为字段

在 Go 语言中,结构体本身也是一种数据类型,所以也可以使用结构体作为字段的类型。例如:

```
type date struct {
    year int
    month int
    day int
}
type student struct {
    id int
    name string
    sex bool
    class string
    birthday date
}
```

该例中首先定义了一个 date 结构体,然后定义结构体 student,date 作为 student 中字段 birthday 的类型。

8.1.2 结构体变量

在 Go 语言中,由于结构体也是一种值类型,所以可以使用它来定义结构体类型的变量,即结构体变量(Variable of the struct type)。用户也可以像使用普通变量一样对结构体变量进行各种各样的操作,比如赋值操作,关系操作等。

1. 结构体变量的声明

当定义了一个结构体之后,就可以使用它来声明结构体变量了。结构体变量的声明和声明普通变量一样,也是使用关键字“var”来进行定义。

例如:

```
var stu student
```

在进行结构体变量声明时,通常也会使用“:=”简写的方式,这种简写方式可以省略关键字“var”。

例如:

```
stu := student{}
```

在使用关键字“var”声明结构体变量时,系统会自动为其分配内存空间。在使用第二种简写方式声明结构体变量时,注意在结构体类型 student 后必须跟一对大括号“{}”,表示声明变量的同时为其初始化存储空间。

2. 字段的访问

声明了结构体变量后,就可以使用点号“.”操作符来访问结构体中的字段了。用户可以使用这种方式直接读字段的值,或者对字段进行赋值。

例如:

```
stu.name = "李明"  
stu.age = 18
```

例 8-1 结构体的定义与简单操作。

```
1 //结构体的定义与简单操作  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7
```



```

8 type student struct {
9     id int
10    name string
11    sex bool
12    age int
13    class string
14 }
15
16 func main() {
17     //标准方式定义
18     var stu1 student
19     //简写方式定义
20     stu2 := student{}
21     stu1.name = "李明"
22     stu2.name = "张衡"
23     stu1.age = 18
24     stu2.age = 19
25     fmt.Println(stu1)
26     fmt.Println(stu2)
27 }

```

编译并运行该程序,输出结果为:

```

{0 李明 false 18 }
{0 张衡 false 19 }

```

分析例 8-1 的运行结果可以看出,除了被赋值的字段 name 和 age 以外,num 字段输出为 0,sex 字段输出为 false,class 字段输出为空。这是因为当定义了一个结构体变量,并且系统为它分配了内存空间以后,结构体中各字段的值将被初始化为该类型的零值。Go 语言规定: int 型的类型零值为 0,bool 型的类型零值为 false,string 型的类型零值为空字符串,所以看到上述的结果也就不奇怪了。

8.1.3 结构体对象

Go 语言中没有类(Class)的概念,但并不表示 Go 不支持面向对象编程,Go 是用结构体来实现面向对象编程模型的。在 Go 语言中,可以使用 new() 函数创建一个结构体对象(Object of the struct type),并为其分配存储空间,调用成功后 new() 函数会返回一个指向该结构体对象的地址指针。

例如:

```

var stu * student
stu = new(student)

```

上例中语句还经常写成下面的简写风格,它们的作用是一致的:

```

stu := new(student)

```

该例中,变量 `stu` 是指向结构体对象(`student`)的一个指针,在 `stu` 的存储空间中,`student` 的各字段都被初始化为该类型的零值。

作为面向对象编程语言的一种惯例,对象中各成员的访问都使用点号“.”操作符。所以,用 `new` 创建的结构体对象中各字段的访问,和使用 `var` 声明的结构体变量中各字段的访问方式相同。

例如:

```
var stu1 student
stu2 := new(student)
stu1.name = "李明"
stu2.name = "张衡"
```

例 8-2 使用 `new()` 函数创建结构体对象。

```
1 //使用 new()函数创建结构体对象
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type date struct {
9     year int
10    month int
11    day int
12 }
13
14 type student struct {
15     Id      int
16     name    string
17     sex     bool
18     class   string
19     birthday date //结构体作为字段
20     _       string //预留字段
21 }
22
23 func main() {
24     stu := new(student)
25     stu.name = "李明"
26     stu.birthday.year = 1995
27     fmt.Println(stu)
28 }
```

编译并运行该程序,输出结果为:

```
&{0 李明 false {1995 0 0}}
```

通过例 8-2 的运行结果可以看出,结构体对象 `stu` 在输出时就是一个指针地址。同样,

除了被赋值的字段外,其他字段都被初始化为类型零值。

在使用 new 来创建一个结构体对象时,它可以出现在程序的任何位置。比如需要在包内声明一个结构体变量,但又不想在程序的最开始为其分配内存,那么这种操作方式就显得特别灵活。

8.1.4 结构体对象初始化

当声明了一个结构体变量或者创建了一个结构体对象后,如果没有初始化,结构体中各字段值将被设置为类型零值,这在前面已经讲过。如果要对结构体中的字段值进行初始化,Go 语言提供了两种方式:全部初始化或部分初始化。

初始化语句通常采用简写方式,格式如下:

1. 初始化全部字段

当要初始化全部字段值时,各字段值按照结构体定义时的顺序依次给出,初始化语句通常采用简写方式,格式如下:

```
structVariableName: = { field11,field12, ... }或  
struct Object Name: = &{ field11,field12, ... }
```

以例 8-1 student 结构体为例:

```
stu1 := student{13001,"李明",false,19,"网络 01"}  
stu2 := &student{13002,"张晓",true,18,"网络 01"}
```

2. 初始化部分字段

如果在初始化时只需要设置部分字段值,那么可以使用“字段名: 字段值”的方式对部分字段进行初始化,格式如下:

```
structVariableName: = { field1Name: field1Value, ... }或  
struct Object Name: = &{ field1Name: field1Value, ... }
```

仍然以例 8-1 为例:

```
stu3 := student{name: "王乐",age: 19}  
stu4 := &student{name: "赵琼",id: 13003}
```

通过该例可以发现,以“字段名: 字段值”进行初始化时,“字段名”可以不依结构体中定义的顺序来出现。比如第二条语句,在 student 结构体中先定义学号 id 字段,然后才定义姓名 name 字段。但在进行部分初始化时,可以不必严格按照这个次序进行。

例 8-3 结构体变量、对象的初始化。

```
1 //结构体变量、对象的初始化  
2 package main
```



```

3
4 import(
5     "fmt"
6 )
7
8 type student struct {
9     Id      int
10    name     string
11    sex      bool
12    age      int
13    class    string
14 }
15
16 func main() {
17     //全部初始化
18     stu1 := student{13001,"李明",false,19,"网络 01"} //结构体变量
19     stu2 := &student{13002,"张晓",true,18,"网络 01"} //结构体对象
20     //部分初始化
21     stu3 := student{name:"王乐",age:19} //结构体变量
22     stu4 := &student{name:"赵琼",id:13003} //结构体对象
23     fmt.Println(stu1)
24     fmt.Println(stu2)
25     fmt.Println(stu3)
26     fmt.Println(stu4)
27 }

```

编译并运行该程序,输出结果为:

```

{13001 李明 false 19 网络 01}
&{13002 张晓 true 18 网络 01}
{0 王乐 false 19 }
&{13003 赵琼 false 0 }

```

8.1.5 结构体的赋值和关系操作

在 Go 语言中,相同类型的结构体之间可以直接使用“=”操作符进行复制赋值。

例如:

```

type user struct {
    id int
    name string
}
func main() {
    user1 := user{100,"张三"}
    fmt.Println(user1)
    var user2 *user = new(user)
    *user2 = user1
    fmt.Println(user2)
}

```

该例测试结果为：

```
{100 张三}  
&{100 张三}
```

从上例执行结果可以看出，“* user2=user1”这条语句执行后，系统将对象 user1 的值复制给了对象 user2。

Go 语言中的结构体除了支持赋值操作符“=”，还支持关系操作符“==”和“!=”，但不支持“>”、“<”等比较操作符。

例如，对于上例执行如下语句：

```
func main() {  
    user1 := user{100, "张三"}  
    user2 := user{200, "李四"}  
    fmt.Println(user1 == user2)  
    fmt.Println(user1 != user2)  
}
```

该例测试结果为：

```
false  
true
```

8.2 嵌入式结构

Go 语言还支持嵌入式结构(Embedded Struct)，所谓嵌入式结构就是没有定义名字的结构体。嵌入式结构可以作为结构体的一个字段，或者可以直接使用嵌入式结构定义结构体变量。

8.2.1 嵌入式结构用作字段

在 8.1.1 节中已经介绍了结构体也可以作为字段类型，这里将说明如何使用嵌入式结构定义字段。使用嵌入式结构作为字段时，嵌入式结构要直接嵌入到外层结构体中。

例如：

```
type student struct {  
    id      int  
    name    string  
    sex     bool  
    class   string  
    birthday struct {    //使用嵌入式结构定义 birthday 字段  
        year int  
        month int  
        day int  
    }  
}
```

该例中,结构体 student 的字段 birthday 就是使用嵌入式结构定义的。

例 8-4 嵌入式结构用作字段。

```
1 //嵌入式结构用作字段
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type student struct {
9     Id      int
10    name     string
11    sex      bool
12    class    string
13    birthday struct {    //使用嵌入式结构定义 birthday 字段
14        year int
15        month int
16        day  int
17    }
18 }
19
20 func main() {
21     stu := new(student)
22     stu.name = "李明"
23     stu.birthday.year = 1995
24     fmt.Println(stu)
25 }
```

编译并运行该程序,输出结果为:

```
&{0 李明 false {1995 0 0}}
```

8.2.2 嵌入式结构直接定义结构体变量

在 Go 语言中,可以不事先声明结构体,而直接使用嵌入式结构定义结构体变量。由于嵌入式结构没有名字,所以使用它定义结构体变量的同时进行初始化。

例如:

```
var usr = struct {
    id      int
    name     string
    password string
}{name: "张三"}
```

该例使用嵌入式结构定义结构体变量 usr,并部分初始化字段 name 的值为“张三”。

在 8.1.3 节中学过 new() 函数可以创建结构体对象,并为其分配存储空间,所以也可以

直接使用 `new()` 函数创建一个嵌入式结构体对象。

例如：

```
var usr = new(struct {
    id      int
    name    string
    password string
})
```

该例中直接使用 `new()` 函数创建了一个结构体对象 `usr`，并为其分配了内存空间，`usr` 的三个字段会被初始化为类型零值。

例 8-5 嵌入式结构直接定义结构体变量以及 `new` 操作。

```
1 //嵌入式结构直接定义结构体变量以及 new 操作
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     //使用嵌入式结构直接定义结构体变量
10    var usr1 = struct {
11        id      int
12        name    string
13        password string
14    }{name: "张三"}
15    fmt.Println(usr1)
16    //使用 new() 函数创建嵌入式结构对象
17    var usr2 = new(struct {
18        id      int
19        name    string
20        password string
21    })
22    usr2.name = "李四"
23    fmt.Println(usr2)
24 }
```

编译并运行该程序，输出结果为：

```
{0 张三 }
&{0 李四 }
```

8.2.3 嵌入式结构直接用于 Map

嵌入式结构除了用作字段、定义变量，它还可以直接用于 `Map`，比如使用嵌入式结构定义 `Map` 的 `Value`。

例如：

```
map1 := map[string]struct {  
    name string  
    age int  
}{  
    "teacher": {"郑智", 39},  
    "student": {"李明", 18},  
}
```

该例中使用嵌入式结构定义 Map 的 Value 类型,这里注意,在使用嵌入式结构定义 Map 时同样要进行初始化工作。

例 8-6 嵌入式结构直接用于 Map。

```
1 //嵌入式结构直接用于 Map  
2 package main  
3  
4 import(  
5     "fmt"  
6 )  
7  
8 func main() {  
9     map1 := map[string]struct {  
10         name string  
11         age int  
12     }{  
13         "teacher": {"郑智", 39},  
14         "student": {"李明", 18},  
15     }  
16     fmt.Println(map1)  
17 }
```

编译并运行该程序,输出结果为:

```
map[teacher:{郑智 39} student:{李明 18}]
```

8.3 匿名字段

在 8.1 节中介绍了结构体的定义方法,定义时是字段名与其类型一一对应。另外,Go 语言还支持只提供类型,而不写字段名的定义方式,这就是匿名字段(Anonymous-field)。匿名字段是将结构体或指针嵌入到另一个结构体中,但不提供字段名,也被称为嵌入字段。需要注意的是,匿名字段的嵌入并不是继承,Go 语言中根本没有继承的概念。

例如：

```
type people struct {  
    name string
```

```
sex bool
}
type teacher struct {
    people    //匿名字段
    department string
}
```

该例首先定义了一个结构体 `people`，然后定义了另外一个结构体 `teacher`。而在定义 `teacher` 时，`people` 直接作为匿名字段嵌入到 `teacher` 中，那么 `teacher` 将包含 `people` 的所有字段。

8.3.1 匿名字段的初始化

Go 语言的匿名字段和大多数面向对象语言 (OOP) 实现继承的方式有些类似，但其本质就是隐式定义了一个以类型为名称的字段。由于是内容嵌入，所以在内存结构上外层结构体和内嵌结构体仍然是一个整体。所以，在初始化结构体成员时，依然要把匿名字段的成员当作一个正常的字段来赋值。

例 8-7 匿名字段的初始化。

```
1 //匿名字段的初始化
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type people struct {
9     name string
10    sex bool
11 }
12
13 type teacher struct {
14     people    //匿名字段
15     department string
16 }
17
18 func main() {
19     //匿名字段成员当作正常字段来赋值
20     teacher1 := teacher{people{"郑智", false}, "Computer Science"}
21     fmt.Println(teacher1)
22 }
```

编译并运行该程序，输出结果为：

```
{郑智 false} Computer Science}
```


8.3.2 匿名字段的访问

Go 语言对点号“.”操作符做了特殊处理,利用它可以直接访问或修改匿名字段成员。

例如对于上例:

```
teacher1.name = "郑智"
teacher1.department = "Yale University"
```

通过该例可以看出,无论是结构体中的正常字段还是匿名字段,都可以直接使用点号“.”操作符来访问,就好像匿名字段就是结构体本身的字段一样。就算是被嵌入匿名字段也包含匿名字段,依然可以用点号“.”操作符直接访问任意嵌入层级的成员,编译器总是从外向里进行查找,直到完成任务或出错。

例 8-8 匿名字段的访问。

```
1 //匿名字段的访问
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type people struct {
9     name string
10    sex bool
11 }
12
13 type teacher struct {
14     people           //匿名字段
15     department string
16 }
17
18 type departmentHead struct {
19     teacher           //有两层嵌入的匿名字段
20     college string
21 }
22
23 func main() {
24     head := departmentHead{}
25     head.name = "郑智"
26     head.department = "Computer Science"
27     head.college = "Yale University"
28     fmt.Println(head)
29 }
```

编译并运行该程序,输出结果为:

```
{{{郑智 false} Computer Science} Yale University}
```

在例 8-8 中,结构体 departmentHead 有两层匿名字段的嵌入,但无论嵌入多少层,匿名字段中的成员访问方法和正常字段一样。Go 语言中,就是利用匿名字段的这种访问特性,实现字段的继承。

8.3.3 匿名字段的多种形式

在 Go 语言中,除了结构体本身,其他所有的内置数据类型和自定义类型都可以作为匿名字段。

例 8-9 其他类型作为匿名字段。

```

1 //其他类型作为匿名字段
2 package main
3
4 import(
5     "fmt"
6 )
7 //自定义类型
8 type skills []string
9
10 type people struct {
11     name string
12     sex bool
13 }
14
15 type teacher struct {
16     people           //struct 作为匿名字段
17     skills           //自定义类型作为匿名字段
18     int             //内置数据类型作为匿名字段
19     department string
20 }
21
22 func main() {
23     teacher1 := teacher{}
24     teacher1.name = "郑智"
25     teacher1.skills = append(teacher1.skills, "Computer", "Golang")
26     teacher1.int = 100
27     teacher1.department = "Computer Science"
28     fmt.Println(teacher1)
29 }
```

编译并运行该程序,输出结果为:

```
{{郑智 false} [Computer Golang] 100 Computer Science}
```

从例 8-9 可以看出,不仅 struct 能作为结构体的匿名字段,自定义类型、内置数据类型都可以作为结构体的匿名字段,而且可以在相应的字段上进行函数操作。例如,本例 skills 是一个自定义的 string 型的 Slice,在程序的第 25 行使用 append() 函数向 skills 中增加元素。

8.3.4 匿名字段的重名

匿名字段的嵌套可能导致在不同层级上出现同名字段,这种情况会导致一些系统隐藏行为的发生。

(1) 外部字段会隐藏匿名字段同名成员。比如对于例 8-7,如果 people 里面有一个 phone 字段,而 teacher 里也有一个 phone 字段,当要访问 phone 时,Go 语言会按照最外层优先的原则。也就是当通过 teacher.phone 访问时,是访问 teacher 里的字段,而不是 people 里的字段。

例如:

```
teacher1 := teacher{people{"郑智", false, "100-201"}, "Computer Science", "200-401"}
fmt.Println(teacher1.phone)
```

输出结果是: 200-401。

这样就允许去重载通过匿名字段继承的一些字段,当然如果想要访问重载后对应匿名字段里的成员,可以通过匿名字段类型前缀来访问。

例如:

```
teacher1 := teacher{people{"郑智", false, "100-201"}, "Computer Science", "200-401"}
fmt.Println(teacher1.people.phone)
```

此时输出结果就是: 100-201。

例 8-10 同名字段的隐藏。

```
1 //同名字段的隐藏
2 package main
3
4 import(
5     "fmt"
6 )
7 type people struct {
8     name string
9     sex bool
10    phone string           //重名字段
11 }
12 type teacher struct {
13     people
14     department string
15     phone    string       //重名字段
16 }
17 func main() {
18     teacher1 := teacher{people{"郑智", false, "100-201"}, "Computer Science", "200-401"}
19     //外部字段隐藏了匿名字段同名成员
20     fmt.Println(teacher1.phone)
21     //通过匿名字段类型前缀访问被嵌入的同名字段
```



```
22     fmt.Println(teacher1.people.phone)
23 }
```

编译并运行该程序,输出结果为:

```
200 - 401
100 - 201
```

(2) 如果多个匿名字段在同一层次重名,将导致编译器无法确定目标而出错。例如,例 8-11 中结构体 d1、d2 中定义了一个 int 型字段 x,而 d1、d2 又最为结构体 data 的嵌入式匿名字段,那么在 data 的内存结构中就存在两个同层级的字段 x。如果进行下述的访问,将导致编译错误。

例如:

```
d := data{d1{10},d2{20}}
fmt.Println(d.x)}
```

编译器在编译时会显示错误信息: ambiguous selector d.x,意思是 d.x 访问歧义,也就是说同层级的同名字段访问会导致编译错误。如果改为匿名字段类型前缀访问,就不会引发编译错误了。

例如:

```
d := data{d1{10},d2{20}}
fmt.Println(d.d1.x,d.d2.x)}
```

例 8-11 通过匿名字段类型前缀访问同层级的同名字段。

```
1 //通过匿名字段类型前缀访问同层级的同名字段
2 package main
3
4 import(
5     "fmt"
6 )
7 type d1 struct {
8     x int
9 }
10 type d2 struct {
11     x int
12 }
13 type data struct {
14     d1
15     d2
16 }
17 func main() {
18     d := data{d1{10},d2{20}}
19     fmt.Println(d.d1.x,d.d2.x)
20 }
```

编译并运行该程序,输出结果为:

```
10 20
```

8.3.5 匿名类型指针

匿名字段还可以作为指针类型嵌入到结构体中,其访问方式和直接嵌入匿名字段相同。

例 8-12 匿名类型指针。

```
1 //匿名类型指针
2 package main
3
4 import(
5     "fmt"
6 )
7 type people struct {
8     name string
9     sex bool
10 }
11 type teacher struct {
12     * people //匿名类型指针
13     department string
14 }
15 func main() {
16     teacher1 := teacher{&people{"郑智",false},"Computer Science"}
17     //匿名类型指针访问和普通匿名字段访问方式相同
18     fmt.Println(teacher1, teacher1.name, teacher1.department)
19 }
```

编译并运行该程序,输出结果为:

```
{0x10e15130 Computer Science} 郑智 Computer Science
```

在例 8-12 的运行输出结果中,“0x10e15130”为匿名类型指针的内存地址。

8.4 方法

Go 语言虽然没有类(Class),但同样支持方法(Method),Go 语言里的 Method 其实就是一个带接收者(Receiver)的函数。

8.4.1 结构化程序设计思想

Go 语言是同时支持结构化和面向对象设计思想的一门静态语言,从某种程度上来讲,用结构化方法开发的软件,其稳定性、可修改性和可重用性都比较差。这是因为结构化方法的本质是功能分解,从代表目标系统整体功能的单个处理着手,自顶向下不断把复杂的处理

分解为子处理,这样一层一层地分解下去,直到只剩下若干个容易实现的子处理功能为止,然后用相应的工具来描述各个最低层的处理。因此,结构化方法是围绕实现处理功能的“过程”来构造系统的。

然而,用户需求的变化大部分是针对“功能”的,因此,这种变化对于基于过程的设计来说是灾难性的。用这种方法设计出来的系统结构常常是不稳定的,用户需求的变化往往造成系统结构的较大变化,从而需要花费很大代价才能实现这种变化。

比如有一个任务,要求定义一个表示矩形的结构体 `rectangle`,它有两个字段:长和宽,然后要求通过矩形的长和宽计算面积,按照结构化程序设计思想,可以按照下面的方式来实现。

```
package main
import (
    "fmt"
)
type rectangle struct {
    width int
    height int
}
func area(r rectangle) int {
    return r.width * r.height
}
func main() {
    r1 := rectangle{4,3}
    r2 := rectangle{30,15}
    fmt.Println(area(r1),area(r2))
}
```

这段代码使用结构化思想解决求面积问题,首先定义结构体 `rectangle`,然后设计了一个求面积函数 `area()`,最后 `rectangle` 的两个实例(对象)`r1,r2` 作为参数传入 `area()` 函数计算面积。

这样实现当然没有问题,但是当需要增加圆形、正方形、五边形甚至其他多边形的时候,该如何计算它们的面积? 只能增加新的函数,但是函数名就必须跟着换了,变成 `area_rectangle`,`area_circle`,`area_triangle` 等。

使用结构化程序设计思想,函数并不属于某个结构体,而是单独存在于结构体外围。这样,函数与结构体之间没有紧密的联系,整个程序呈现出一种松散状态。在这种状态下,用户在阅读程序时将变得非常吃力,也不利于程序的后期维护与升级。

8.4.2 面向对象程序设计思想

基于上述原因,Go 语言也支持面向对象程序设计思想。面向对象(Object Oriented, OO)就是将要研究的事物抽象成对象,然后对对象的状态和行为进行定义。

1. 对象

在 Go 语言里,对象(Object)可以是最简单的内置数据类型,或者是复杂的结构体。对象不仅能表示具体的事物,还能表示抽象的规则、计划或事件。比如在 8.4.1 节的例子中,

结构体 `rectangle` 就是一个定义了矩形长和宽的抽象对象。之所以说它抽象,是因为它本身并不表示任何一个具体存在矩形,只是对矩形进行了定义。

2. 对象的状态

每一个对象都具有状态,对象通常用数值来描述它的状态。比如在 8.4.1 节的例子中,`r1`,`r2` 就是两个拥有具体长和宽数值的矩形,它们是结构体 `rectangle` 的两个实例对象,`rectangle` 是抽象的,而 `r1`,`r2` 是具体存在的。

3. 对象的操作

对象还应有操作,用于改变对象的状态,操作就是对象的行为。比如在 8.4.1 节的例子中,函数 `area()` 就是对象 `r1`,`r2` 的操作,可以计算 `r1`,`r2` 的面积。但是采取结构化程序设计思想,函数 `area()` 和对象 `rectangle` 之间没有必然的联系性,所以函数 `area()` 不是真正意义上对象的一个操作。

在 Go 语言中,对对象的操作称为 Method(方法),Method 就是在函数前增加了一个接收者(Receiver)对象。这样,操作就和对象真正关联起来了。

例如,对 8.4.1 节例子中的代码做如下修改,函数 `area()` 就会变成对象 `rectangle` 的操作方法了。

```
package main
import (
    "fmt"
)
type rectangle struct {
    width int
    height int
}
func (recv rectangle) area() int {
    return recv.width * recv.height
}
func main() {
    r1 := rectangle{4,3}
    r2 := rectangle{30,15}
    fmt.Println(r1.area(),r2.area())
}
```

面向对象程序设计思想实现了数据和操作的结合,从而也将对象和操作组织成为一个有机的整体。在阅读代码时,哪个方法作用于哪个对象一目了然,就不至于产生混淆了。当然,采用面向对象的程序设计思想,程序的整体结构、设计思路也会变得特别清晰。尤其是对一些大型的、结构复杂的程序来说,采用面向对象的设计方法就显得尤为重要了。

8.4.3 Method 的基本定义

Go 语言的 Method 类似于一个函数,只是函数名前多了个绑定类型参数——receiver。Method 的基本定义格式如下:

```
func (recv receiver_type) methodName(参数列表)(返回值){
    :
}
```

Method 中的 Receiver 可以是内置类型、自定义类型、结构体或指针类型。

例 8-13 Method 的基本定义。

```
1 //Method 的基本定义
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type rectangle struct {
9     width int
10    height int
11 }
12 //函数 area()是对象 rectangle 的一个方法
14 func (recv rectangle) area() int {
15     return recv.width * recv.height
16 }
17 func main() {
18     r1 := rectangle{4,3}
19     r2 := rectangle{30,15}
20     fmt.Println(r1.area(),r2.area())
21 }
```

编译并运行该程序,输出结果为:

```
12 450
```

8.4.4 多个 Method 可以同名

(1) 在定义 Method 时,多个 Method 可以同名,但如果接收者不一样,那 Method 就不一样。在例 8-14 中定义了两个结构体对象 rectangle 和 circle,它们有一个同名的方法 area()用来计算各自对象的面积。

例 8-14 多个 Method 可以同名。

```
1 //多个 Method 可以同名
2 package main
3
4 import(
5     "fmt"
6     "math"
7 )
8
```

```
9 type rectangle struct {
10     width int
11     height int
12 }
13 type circle struct {
14     radius float32
15 }
16 //此处 area()是对象 rectangle 的一个方法
17 func (recv rectangle) area() int {
18     return recv.width * recv.height
19 }
20 //此处 area()是对象 circle 的一个方法
21 func (recv circle) area() float32 {
22     return recv.radius * recv.radius * math.Pi
23 }
24 func main() {
25     r1 := rectangle{4,3}
26     c1 := circle{5}
27     fmt.Println(r1.area(),c1.area())
28 }
```

编译并运行该程序,输出结果为:

```
12 78.53982
```

(2) 如果普通类型作为 Receiver,它只是一个值传递;而指针类型作为 Receiver,它将是一个引用传递。两者的差别在于,指针作为 Receiver 会对实例对象的内容发生操作,而普通类型作为 Receiver 仅是以副本作为操作对象,并不对原实例对象发生操作,本章后面内容对此会有详细论述。

(3) Method 的名字可能一样,但如果接收者不一样,那 Method 就不一样。

(4) Method 里面可以访问接收者的字段,调用 Method 进行访问,就像在 Struct 里访问字段一样。

8.4.5 指针作为 Receiver

如果普通类型作为 Receiver,它只是一个值传递;而指针类型作为 Receiver,它将是一个引用传递。两者的差别在于,指针作为 Receiver 会对实例对象的内容发生操作,而普通类型作为 Receiver 仅是以副本作为操作对象,并不对原实例对象发生操作。

例如,下例定义了一个结构体对象 coordinate,它用于记录平面坐标值 x,y,方法 swap()的作用是交换坐标值 x,y,首先使用普通类型作为 Receiver,观察程序运行结果。

```
package main
import (
    "fmt"
)
```



```

type coordinate struct {
    x int
    y int
}

func (recv coordinate) swap() {
    var temp int
    temp, recv.x, recv.y = temp, recv.y, recv.x
    fmt.Println(recv)
}

func main() {
    r1 := coordinate{3,4}
    r1.swap()
    fmt.Println(r1)
}

```

上例测试结果为：

```

{4 3}
{3 4}

```

通过程序运行结果可以发现，普通类型作为 Receiver，它只是一个值传递，在方法 swap() 中对对象属性值的操作并不会改变原实例的值。

如果使用指针作为 Receiver，对上例代码进行修改，再观察程序运行结果。

```

func (recv *coordinate) swap() {
    var temp int
    temp, recv.x, recv.y = temp, recv.y, recv.x
    fmt.Println(recv)
}

func main() {
    r1 := coordinate{3,4}
    p := &r1
    p.swap()
    fmt.Println(r1)
}

```

上例测试结果为：

```

&{4 3}
{4 3}

```

通过程序运行结果可以发现，指针类型作为 Receiver，它将是一个引用传递，引用传递复制的是对象的内存地址，所以在方法 swap() 中对对象属性值的操作将直接影响到原实例的值。

和其他面向对象语言相比，在 Go 语言中没有隐藏的 this 指针。而在 C++、Java、C# 这类语言中，它们的成员方法中都带有一个隐藏的 this 指针，而在 Python 语言中会隐藏一个 self 参数，它和其他语言的 this 作用一样。

对于 Go 语言来说,方法作用的 Receiver 是显式传递,没有被隐藏起来,所有操作对用户都是可见的。另外,方法作用的 Receiver 不一定必须是指针,也不用必须叫做 this,比如叫做 recv、r 等都可以。

8.4.6 匿名 Receiver

如果方法代码中从不使用 Receiver 参数,那么就可以省略 Receiver 的变量名,此时的接收者将是一个匿名 Receiver。

例 8-15 匿名 Receiver。

```
1 //匿名 Receiver
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type object struct {
9     id int
10    name string
11 }
12 func (object) msgbox() {
13     fmt.Println("This is a object!")
14 }
15 func (*object) msgBox() {
16     fmt.Println("This is a object!")
17 }
18 func main() {
19     obj := object{}
20     p := &obj
21     obj.msgbox()
22     p.msgBox()
23 }
```

编译并运行该程序,输出结果为:

```
This is a object!
This is a object!
```

在例 8-15 中为对象 object 定义了两个匿名 Receiver,一个是普通类型,另外一个是指针类型。注意,这种情况下方法名不能同名,要不然编译会出错。该例中普通类型 Receiver 的方法名为 msgbox(),指针类型 Receiver 的方法名为 msgBox()。

8.4.7 Method 的继承

通过前面 8.3 节学习了在 Go 语言中,通过匿名字段可以实现字段继承,在本节中还会发现 Go 语言的一个神奇之处,即 Method 也是可以继承的。如果匿名字段实现了一个

Method,那么包含这个匿名字段的 Struct 对象也能调用该 Method。

例 8-16 Method 的继承。

```
1 //Method 的继承
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type people struct {
9     name string
10    phone string
11 }
12 type teacher struct {
13     people
14     department string
15 }
16 type student struct {
17     people
18     school string
19 }
20 func (r people) sayHi() {
21     fmt.Printf("Hi,I'm %s you can call me on %s.\n",r.name,r.phone)
22 }
23 func main() {
24     teacher1 := teacher{people{"郑智","010-22002"},"Computer Science"}
25     student1 := student{people{"李明","010-11001"},"Yale University"}
26     teacher1.sayHi()
27     student1.sayHi()
28 }
```

编译并运行该程序,输出结果为:

```
Hi,I'm 郑智 you can call me on 010-22002.
Hi,I'm 李明 you can call me on 010-11001.
```

在例 8-16 中,people 作为 teacher 和 student 的匿名字段,sayHi()是为 people 实现的一个方法。作为外层结构,teacher 和 student 同时也继承了 people 所拥有的方法,所以在 teacher 和 student 的对象中可以直接调用方法 sayHi()。

8.4.8 Method 的重写

在例 8-16 中,如果 student 想要实现自己的 sayHi()方法,可以采用 Method 重写的方法来解决。这和匿名字段重名是一样的道理,即外部方法会隐藏匿名字段同名方法。所以可以在 student 上再定义一个 Method,重写了匿名字段的方法。

例 8-17 Method 的重写。

```
1 //Method 的重写
2 package main
3
4 import(
5     "fmt"
6 )
7 type people struct {
8     name string
9     phone string
10 }
11 type teacher struct {
12     people
13     department string
14 }
15 type student struct {
16     people
17     school string
18 }
19 func (r people) sayHi() {
20     fmt.Printf("Hi, I'm %s you can call me on %s.\n", r.name, r.phone)
21 }
22 func (s student) sayHi() {
23     fmt.Printf("Hi, I'm %s. I study in %s, call me on %s.\n", s.name, s.school, s.phone)
24 }
25 func main() {
26     teacher1 := teacher{people{"郑智", "010-22002"}, "Computer Science"}
27     student1 := student{people{"李明", "010-11001"}, "Yale University"}
28     teacher1.sayHi()
29     student1.sayHi()
30 }
```

编译并运行该程序,输出结果为:

```
Hi, I'm 郑智 you can call me on 010-22002.
Hi, I'm 李明. I study in Yale University, call me on 010-11001.
```

通过例 8-17 的运行结果可以看出,对象 student1 在调用 sayHi() 方法时,Receiver 为 people 的方法将被隐藏,实际调用的是 Receiver 为 student 的这个外部 sayHi() 方法。

8.5 可见性规则和 Struct 的导入

和调用函数一样(见 7.2.3 节),有时也需要从外部包(Exterior Package)导入一个 Struct,那么在定义外部 Struct 时也要遵循可见性规则。

8.5.1 可见性规则

Go 语言对关键字的数量严格控制,其中并没有像其他面向对象语言中的 `private`、`protected`、`public` 这样的关键字。要使包内的某个标识符对其他包可见,该标识符的首写字母必须大写,这就是 Go 语言的可见性规则,它的作用和其他语言的 `public` 关键字一样。

例如:

```
type User struct {
    Id int
    Name string
    age int
}
func (u * User) SayHi(){
    fmt.Printf("Hi, I'm %s. Nice to meet you!\n", u.Name)
}
func (u User) old() int{
    return u.age
}
```

该例中结构体对象 `User` 首写字母是大写,就可以被其他包引用。`User` 中字段 `Id`、`Name` 对外部包是可见的,而 `age` 字段对外部包不可见。对 `User` 成员方法的访问也遵循可见性规则,所以外部包可以访问方法 `SayHi()`,而不能访问方法 `old()`。

8.5.2 Struct 的导入

要使用一个外部 Struct,首先要导入外部包,格式如下:

```
import "./pack_directory/pack_name"
```

上式中“`pack_directory`”是包文件所存放的文件目录,“`pack_name`”是外部 Struct 所存在的包名。

导入外部包后,就可以引用并访问外部包了,格式如下:

```
pack_name.Struct_name
```

上式中“`pack_name`”是包名,“`Struct_name`”是外部结构体名,注意按照可见性规则,首写字母是大写的外部 Struct 才可能被引用。

比如 `main.go` 中有一个 `User` 结构体对象,它是来自于 `mystruct` 包,此时需首先构建 `mystruct` 包,然后才能引用 `User` 对象。

构建 `mystruct` 包步骤如下:

- (1) 建立 `mystruct.go` 源文件。
- (2) 编辑 `mystruct.go` 文件,添加源代码,如下所示。

```

/*
    mystruct 包定义了 User 对象,并实现了两个对 User 进行操作的方法.
    SayHi()方法
    old()方法
*/
package mystruct

import (
    "fmt"
)

type User struct {
    Id int
    Name string
    age int
}

func (u * User) SayHi() {
    fmt.Printf("Hi, I'm %s. Nice to meet you!\n", u.Name)
}

func (u User) old() int {
    return u.age
}

```

(3) 执行如下命令:

```
go install mystruct
```

如果 GOPATH 变量配置正确,则会在 GOPATH 的 pkg 目录下生成 mystruct.a 包文件,要导入 mystruct 包执行如下命令即可。

```
import "mystruct"
```

例 8-18 Struct 的导入。

```

1 //Struct 的导入
2 package main
3
4 import(
5     "fmt"
6     "mystruct"
7 )
8 func main() {
9     usr := new(mystruct.User)
10    usr.Id = 100
11    usr.Name = "张三"
12    usr.SayHi()

```



```
13     fmt.Println(usr)
14 }
```

编译并运行该程序,输出结果为:

```
Hi,I'm 张三. Nice to meet you!
&{100 张三 0}
```

在 main.go 中引用并使用 User 对象会发现,mystruct 包中定义的 age 字段,old() 方法对 main 包是不可见的。

8.6 字段标签

Go 语言允许为字段定义标签(Tag),Struct 中的不同字段不仅可以通过字段名和类型区分,还可以通过字段标签(Field-tag)来区分。字段标签通常用于程序使用手册的书写,或者作为一些重要的标记符。

在 Go 语言中,不允许在普通程序语句中访问字段标签,字段标签只能使用反射包(Reflect)中提供的特殊方法进行读取,利用 Reflect 可以获取字段名、字段类型和字段值。如果是 Struct 类型,Reflect 就可以使用字段名作为索引读取字段标签。

例 8-19 字段标签。

```
1 //字段标签
2 package main
3
4 import(
5     "fmt"
6     "reflect"
7 )
8 type user struct {
9     Id    int    "账号"
10    Name string "姓名"
11    Sex   bool   "性别"
12 func main() {
13     u := user{100,"张三",false}
14     //使用 TypeOf()函数获取对象的类型
15     t := reflect.TypeOf(u)
16     //使用 ValueOf()获取对象的值
17     v := reflect.ValueOf(u)
18     for i := 0; i < t.NumField(); i++{
19         f := t.Field(i)
20         fmt.Printf("%s( %s = %v)\n",f.Tag,f.Name,v.Field(i).Interface())
21     }
22 }
```

编译并运行该程序,输出结果为:

```
账号 (Id = 100)
姓名 (Name = 张三)
性别 (Sex = false)
```

8.7 数据 I/O 对象及操作

Go 语言标准库 bufio 包,实现了对数据 I/O 接口的缓冲功能。它封装于接口 io. Reader、io. Reader 和 io. Writer 中,并对应创建对象 ReadWriter、Reader 或 Writer,在提供缓冲的同时实现了一些文本基本 I/O 操作功能。

8.7.1 ReadWriter 对象

ReadWriter 对象可以对数据 I/O 接口 io. ReadWriter 进行输入输出缓冲操作,ReadWriter 结构定义如下:

```
type ReadWriter struct {
    * Reader
    * Writer
}
```

默认情况下,ReadWriter 对象中存放了一对 Reader 和 Writer 指针,它同时提供了对数据 I/O 对象的读写缓冲功能。

可以使用 NewReadWriter() 函数创建 ReadWriter 对象,该函数的功能是根据指定的 Reader 和 Writer 创建一个 ReadWriter 对象,ReadWriter 对象将会向底层 io. ReadWriter 接口写入数据,或者从 io. ReadWriter 接口读取数据。该函数原型声明如下:

```
func NewReadWriter(r * Reader, w * Writer) * ReadWriter
```

在函数 NewReadWriter() 中,参数 r 是要读取的来源 Reader 对象;参数 w 是要写入的目的 Writer 对象。

8.7.2 Reader 对象

Reader 对象可以对数据 I/O 接口 io. Reader 进行输入缓冲操作,Reader 结构定义如下:

```
type Reader struct {
    //contains filtered or unexported fields
}
```

默认情况下 Reader 对象没有定义初始值,输入缓冲区最小值为 16。当超出限制时,另创建一个二倍的存储空间。

1. Reader 对象创建函数

“Reader 对象的创建函数共有 2 个：NewReader() 和 NewReaderSize(), 下面分别介绍。”

(1) NewReader() 函数。NewReader() 函数的功能是按照缓冲区默认长度创建 Reader 对象, Reader 对象会从底层 io.Reader 接口读取尽量多的数据进行缓存。该函数原型声明如下:

```
func NewReader(rd io.Reader) * Reader
```

在函数 NewReader() 中, 参数 rd 是 io.Reader 接口, Reader 对象将从该接口读取数据。

(2) NewReaderSize() 函数。NewReaderSize() 函数的功能是按照指定的缓冲区长度创建 Reader 对象, Reader 对象会从底层 io.Reader 接口读取尽量多的数据进行缓存。该函数原型声明如下:

```
func NewReaderSize(rd io.Reader, size int) * Reader
```

在函数 NewReaderSize() 中, 参数 rd 是 io.Reader 接口; 参数 size 是指定的缓冲区字节长度。

2. Reader 对象操作方法

Reader 对象的操作方法共有 11 个: Read()、ReadByte()、ReadBytes()、ReadLine()、ReadRune()、ReadSlice()、ReadString()、UnreadByte()、UnreadRune()、Buffered() 和 Peek(), 下面分别介绍。

(1) Read() 方法。Read() 方法的功能是读取数据, 并存放到字节切片 p 中。Read() 执行结束会返回已读取的字节数, 因为最多只调用底层的 io.Reader 一次, 所以返回的 n 可能小于 len(p); 当字节流结束时, n 为 0, err 为 io.EOF。该方法原型声明如下:

```
func (b * Reader) Read(p []byte) (n int, err error)
```

在方法 Read() 中, 参数 p 是用于存放读取数据的字节切片。

例如:

```
func main() {  
    data := []byte("中华人民共和国")  
    rd := bytes.NewReader(data)  
    r := bufio.NewReader(rd)  
    var buf [128]byte  
    n, err := r.Read(buf[:])  
    fmt.Println(string(buf[:n]), n, err)  
}
```

该例测试结果为:

```
中华人民共和国 21 <nil>
```


(2) `ReadByte()`方法。`ReadByte()`方法的功能是读取并返回一个字节。如果没有字节可读,则返回错误信息。该方法原型声明如下:

```
func (b * Reader) ReadByte() (c byte, err error)
```

例如:

```
func main() {  
    data := []byte("Golang")  
    rd := bytes.NewReader(data)  
    r := bufio.NewReader(rd)  
    c, err := r.ReadByte()  
    fmt.Println(string(c), err)  
}
```

该例测试结果为:

```
G<nil>
```

(3) `ReadBytes()`方法。`ReadBytes()`方法的功能是 `ReadBytes` 读取数据直到遇到第一个分隔符“`delim`”,并返回读取的字节序列(包括“`delim`”)。如果 `ReadBytes` 在读到第一个“`delim`”之前出错,它返回已读取的数据和那个错误(通常是 `io.EOF`)。只有当返回的数据不以“`delim`”结尾时,返回的 `err` 才不为空值。该方法原型声明如下:

```
func (b * Reader) ReadBytes(delim byte) (line []byte, err error)
```

在方法 `ReadBytes()`中,参数 `delim` 用于指定分割字节。

例如:

```
func main() {  
    data := []byte("Hello, world!")  
    rd := bytes.NewReader(data)  
    r := bufio.NewReader(rd)  
    var delim byte = ','  
    line, err := r.ReadBytes(delim)  
    fmt.Println(string(line), err)  
}
```

该例测试结果为:

```
Hello,<nil>
```

(4) `ReadLine()`方法。`ReadLine()`方法是一个低级的用于读取一行数据的原语,大多数调用者应该使用 `ReadBytes('\n')`或者 `ReadString('\n')`。`ReadLine` 试图返回一行,不包括结尾的回车字符。如果一行太长了(超过缓冲区长度),参数 `isPrefix` 会设置为 `true` 并且只返回前面的数据,剩余的数据会在以后的调用中返回。当返回最后一行数据时,参数

isPrefix 会置为 false。返回的字节切片只在下一次调用 ReadLine 前有效。ReadLine 或者返回一个非空的字节切片或者返回一个错误,但它们不会同时返回。该方法原型声明如下:

```
func (b * Reader)ReadLine()(line []byte, isPrefix bool, err error)
```

例如:

```
func main() {
    data := []byte("Golang is a beautiful language.
                    \r\n I like it!")
    rd := bytes.NewReader(data)
    r := bufio.NewReader(rd)
    line, prefix, err := r.ReadLine()
    fmt.Println(string(line), prefix, err)
}
```

该例测试结果为:

```
Golang is a beautiful language. false <nil>
```

(5) ReadRune()方法。ReadRune()方法的功能是读取一个 UTF-8 编码的字符,并返回其 Unicode 编码和字节数。如果编码错误,ReadRune 只读取一个字节并返回 unicode.ReplacementChar(U+FFFD)和长度 1。该方法原型声明如下:

```
func (b * Reader) ReadRune() (r rune, size int, err error)
```

例如:

```
func main() {
    data := []byte("中华人民共和国")
    rd := bytes.NewReader(data)
    r := bufio.NewReader(rd)
    ch, size, err := r.ReadRune()
    fmt.Println(string(ch), size, err)
}
```

该例测试结果为:

```
中 3 <nil>
```

(6) ReadSlice()方法。ReadSlice()方法的功能是读取数据直到分隔符“delim”处,并返回读取数据的字节切片,下次读取数据时返回的切片会失效。如果 ReadSlice 在查找到“delim”之前遇到错误,它返回读取的所有数据和那个错误(通常是 io.EOF)。如果缓冲区满时也没有查找到“delim”,则返回 ErrBufferFull 错误。因为 ReadSlice 返回的数据会在下次 I/O 操作时被覆盖,大多数调用者应该使用 ReadBytes 或者 ReadString。只有当 line 不以“delim”结尾时,ReadSlice 才会返回非空 err。该方法原型声明如下:

```
func (b * Reader) ReadSlice(delim byte) (line []byte, err error)
```

在方法 ReadSlice() 中, 参数 delim 用于指定分割字节。

例如:

```
func main() {  
    data := []byte("apple,banana,pear")  
    rd := bytes.NewReader(data)  
    r := bufio.NewReader(rd)  
    var delim byte = ','  
    line, err := r.ReadSlice(delim)  
    fmt.Println(string(line), err)  
    line, err = r.ReadSlice(delim)  
    fmt.Println(string(line), err)  
    line, err = r.ReadSlice(delim)  
    fmt.Println(string(line), err)  
}
```

该例测试结果为:

```
apple,<nil>  
banana,<nil>  
pear EOF
```

(7) ReadString() 方法。ReadString() 方法的功能是读取数据直到分隔符“delim”第一次出现, 并返回一个包含“delim”的字符串。如果 ReadString 在读取到“delim”前遇到错误, 它返回已读字符串和那个错误(通常是 io.EOF)。只有当返回的字符串不以“delim”结尾时, ReadString 才返回非空 err。该方法原型声明如下:

```
func (b * Reader) ReadString(delim byte) (line string, err error)
```

在方法 ReadString() 中, 参数 delim 用于指定分割字节。

例如:

```
func main() {  
    data := []byte("apple,banana,pear")  
    rd := bytes.NewReader(data)  
    r := bufio.NewReader(rd)  
    var delim byte = ','  
    line, err := r.ReadString(delim)  
    fmt.Println(line, err)  
}
```

该例测试结果为:

```
apple,<nil>
```


(8) `UnreadByte()`方法。`UnreadByte()`方法的功能是 `UnreadByte` 取消已读取的最后一个字节(即把字节重新放回读取缓冲区的前部)。只有最近一次读取的单个字节才能取消读取。该方法原型声明如下:

```
func (b * Reader) UnreadByte() error
```

(9) `UnreadRune()`方法。`UnreadRune()`方法的功能是 `UnreadRune` 取消读取最后一次读取的 Unicode 字符。如果最后一次读取操作不是 `ReadRune`, `UnreadRune` 会返回一个错误(在这方面它比 `UnreadByte` 更严格,因为 `UnreadByte` 会取消上次任意读操作的最后一个字节)。该方法原型声明如下:

```
func (b * Reader) UnreadRune() error
```

(10) `Buffered()`方法。`Buffered()`方法的功能是返回可从缓冲区读出数据的字节数,该方法原型声明如下:

```
func (b * Reader) Buffered() int
func main() {
    data := []byte("中华人民共和国")
    rd := bytes.NewReader(data)
    r := bufio.NewReader(rd)
    var buf [12]byte
    n, err := r.Read(buf[:])    //第一次执行 Read 读取 4 个 Unicode 字符
    fmt.Println(string(buf[:n]), n, err)
    rn := r.Buffered()
    fmt.Println(rn)
    n, err = r.Read(buf[:])    /第二次读取了剩余的 3 个 Unicode 字符
    fmt.Println(string(buf[:n]), n, err)
    rn = r.Buffered()
    fmt.Println(rn)
}
```

该例测试结果为:

```
中华人民 12 <nil>
还可从缓冲区读取 9 个字节
共和国 9 <nil>
还可从缓冲区读取 0 个字节
```

(11) `Peek()`方法。`Peek()`方法的功能是读取指定字节数的数据,这些被读取的数据不会从缓冲区中清除。在下次读取之后,本次返回的字节切片会失效。如果 `Peek` 返回的字节数不足 `n` 字节,则会同时返回一个错误说明原因;如果 `n` 比缓冲区要大,则错误为 `ErrBufferFull`。该方法原型声明如下:

```
func (b * Reader) Peek(n int) ([]byte, error)
```

在方法 `Peek()` 中, 参数 `n` 是希望读取的字节数。

例如:

```
func main() {
    data := []byte("中华人民共和国")
    rd := bytes.NewReader(data)
    r := bufio.NewReader(rd)
    b1, err := r.Peek(9)
    fmt.Println(string(b1), err)
    b1, err = r.Peek(18)
    fmt.Println(string(b1), err)
    b1, err = r.Peek(27)
    fmt.Println(string(b1), err)
}
```

该例测试结果为:

```
中华人 <nil>
中华人民共和 <nil>
中华人民共和国 EOF
```

8.7.3 Writer 对象

Writer 对象可以对数据 I/O 接口 `io.Writer` 进行输出缓冲操作, Writer 结构定义如下:

```
type Writer struct {
    //contains filtered or unexported fields
}
```

默认情况下 Writer 对象没有定义初始值, 如果输出缓冲过程中发生错误, 则数据写入操作立刻被终止, 后续的写操作都会返回写入异常错误。

1. Writer 对象创建函数

“Writer 对象的创建函数共有 2 个: `NewWriter()` 和 `NewWriterSize()`, 下面分别介绍。”

(1) `NewWriter()` 函数。 `NewWriter()` 函数的功能是按照默认缓冲区长度创建 Writer 对象, Writer 对象会将缓存的数据批量写入底层 `io.Writer` 接口。该函数原型声明如下:

```
func NewWriter(wr io.Writer) * Writer
```

在函数 `NewWriter()` 中, 参数 `wr` 是 `io.Writer` 接口, Writer 对象会将数据写入该接口。

(2) `NewWriterSize()` 函数。 `NewWriterSize()` 函数的功能是按照指定的缓冲区长度创建 Writer 对象, Writer 对象会将缓存的数据批量写入底层 `io.Writer` 接口。该函数原型声明如下:

```
func NewWriterSize(wr io.Writer, size int) * Writer
```

在函数 `NewWriterSize()` 中, 参数 `wr` 是 `io.Writer` 接口; 参数 `size` 是指定的缓冲区字节长度。(即把字节重新放回读取缓冲区的前部)。只有最近一次读取的单个字节被清除。

2. Writer 对象操作方法

Writer 对象的操作方法共有 7 个: `Available()`、`Buffered()`、`Flush()`、`Write()`、`WriteByte()`、`WriteRune()` 和 `WriteString()` 方法, 下面分别介绍。

(1) `Available()` 方法。`Available()` 方法的功能是返回缓冲区中未使用的字节数, 该方法原型声明如下:

```
func (b * Writer) Available() int
```

例如:

```
func main() {
    wr := bytes.NewBuffer(nil)
    w := bufio.NewWriter(wr)
    p := []byte("Hello, world!")
    fmt.Println("写入前未使用的缓冲区为:", w.Available())
    w.Write(p)
    fmt.Printf("写入 %q 后, 未使用的缓冲区为: %d\n", string(p),
        w.Available())
}
```

该例测试结果为:

```
写入前未使用的缓冲区为: 4096
写入 "Hello, world!" 后, 未使用的缓冲区为: 4084
```

(2) `Buffered()` 方法。`Buffered()` 方法的功能是返回已写入当前缓冲区中的字节数, 该方法原型声明如下:

```
func (b * Writer) Buffered() int
```

例如:

```
func main() {
    wr := bytes.NewBuffer(nil)
    w := bufio.NewWriter(wr)
    p := []byte("Hello, world!")
    fmt.Println("写入前写入字节数为:", w.Buffered())
    w.Write(p)
    fmt.Printf("写入 %q 后, 写入的字节数为: %d\n", string(p),
        w.Buffered())
    w.Flush()
    fmt.Println("执行 Flush 方法后, 写入的字节数为:", w.Buffered())
}
```


该例测试结果为：

写入前写入字节数为：0

写入"Hello,world!"后,写入的字节数为:12

执行 Flush 方法后,写入的字节数为：0

(3) Flush()方法。Flush()方法的功能是把缓冲区中的数据写入底层的 io. Writer,并返回错误信息。如果成功写入,error 返回 nil; 否则,error 返回错误原因。该方法原型声明如下:

```
func (b * Writer) Flush() error
```

例如:

```
func main() {  
    wr := bytes.NewBuffer(nil)  
    w := bufio.NewWriter(wr)  
    p := []byte("Hello,world!")  
    w.Write(p)  
    fmt.Printf("未执行 Flush 缓冲区输出 %q.\n", string(wr.Bytes()))  
    w.Flush()  
    fmt.Printf("执行 Flush 后缓冲区输出 %q.\n", string(wr.Bytes()))  
}
```

该例测试结果为:

未执行 Flush 缓冲区输出"".

执行 Flush 后缓冲区输出"Hello,world!".

(4) Write()方法。Write()方法的功能是把字节切片 p 写入缓冲区,返回已写入的字节数 nn。如果 nn 小于 len(p),则同时返回一个错误原因。该方法原型声明如下:

```
func (b * Writer) Write(p []byte) (nn int,err error)
```

在方法 Write()中,参数 p 是要写入的字节切片。

例如:

```
func main() {  
    wr := bytes.NewBuffer(nil)  
    w := bufio.NewWriter(wr)  
    p := []byte("Hello,world!")  
    n,err := w.Write(p)  
    w.Flush()  
    fmt.Println(string(wr.Bytes()),n,err)  
}
```

该例测试结果为:

```
Hello,world! 12 <nil>
```

(5) WriteByte()方法。WriteByte()方法的功能是写入一个字节,如果成功写入,error 返回 nil; 否则,error 返回错误原因。该方法原型声明如下:

```
func (b *Writer) WriteByte(c byte) error
```

在方法 WriteByte()中,参数 c 是要写入的字节数据,比如 ASCII 字符。

例如:

```
func main() {  
    wr := bytes.NewBuffer(nil)  
    w := bufio.NewWriter(wr)  
    var c byte = 'G'  
    err := w.WriteByte(c)  
    w.Flush()  
    fmt.Println(string(wr.Bytes()),err)  
}
```

该例测试结果为:

```
G <nil>
```

(6) WriteRune()方法。WriteRune()方法的功能是以 UTF-8 编码写入一个 Unicode 字符,返回写入的字节数和错误信息。该方法原型声明如下:

```
func (b *Writer) WriteRune(r rune) (size int,err error)
```

在方法 WriteRune()中,参数 r 是要写入的 Unicode 字符。

例如:

```
func main() {  
    wr := bytes.NewBuffer(nil)  
    w := bufio.NewWriter(wr)  
    var r rune = '中'  
    size,err := w.WriteRune(r)  
    w.Flush()  
    fmt.Println(string(wr.Bytes()),size,err)  
}
```

该例测试结果为:

```
中 3 <nil>
```

(7) WriteString()方法。WriteString()方法的功能是写入一个字符串,并返回写入的字节数和错误信息。如果返回的字节数小于 len(s),则同时返回一个错误说明原因。该方

法原型声明如下：

```
func (b * Writer) WriteString(s string) (int,error)
```

在方法 WriteString()中,参数 s 是要写入的字符串。

例如：

```
func main() {
    wr := bytes.NewBuffer(nil)
    w := bufio.NewWriter(wr)
    s := "Hello,world!"
    n,err := w.WriteString(s)
    w.Flush()
    fmt.Println(string(wr.Bytes()),n,err)
}
```

该例测试结果为：

```
Hello,world! 12 <nil>
```

8.8 应用举例——链表操作

有时可以让 Struct 的一个指针成员指向它自己,利用这种特性 Struct 对象可以作为链表或者二叉树的元素,通常叫做节点(Node)。

8.8.1 链表简介

链表是一种常见的重要数据结构,它的主要特点是能动态地进行存储分配。通过第 6 章的学习知道,使用数组存放数据时,必须事先分配固定长度的存储空间(即元素个数)。比如,有的班级有 60 名学生,而有的班级只有 30 名学生,如果要用同一个数组先后存放不同班级的学生信息,则数组长度至少为 60。如果班级人数难以确定,则必须把数组定义得足够大,显然这将会浪费内存。

链表则没有这种缺点,它会根据需要动态开辟内存单元,不会造成内存的浪费。另外,链表还可以动态增添新节点、删除旧节点。图 8-2 表示的是一种最简单的单向链表的结构。

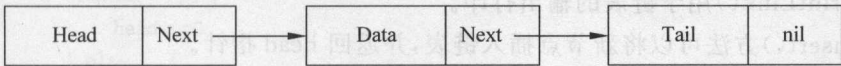


图 8-2 单向链表

单向链表中有一个头指针变量,图中以 Head 表示,头指针存放链表第一个元素的地址。链表中每一个元素称为“节点”,每个节点都应包括两个部分:用户数据和下一个节点的地址。从图 8-2 可以看出,Head 指向第一个元素,第一个元素又指向第二个元素……直到最后一个元素 Tail。Tail 不再指向其他元素,它称为“表尾”,它的地址部分是“nil”,表示

没有,链表到此结束。

可以看出,链表各元素在内存中可以不连续存放。在链表中要找到一个元素,必须先找到上一个元素,根据它提供的 Next 地址才能找到要找的目标。如果不提供“头指针”Head,则整个链表都无法访问。另外,链表节点之间必须一环扣一环,中间不能断开。否则,链表将丢失节点、不完善。

通过上面的介绍可以看到,链表这种数据结构,必须利用指针变量才能实现,即一个节点中应包含一个指针变量,用它存放下一个节点的地址。

8.8.2 Struct 和 Method 设计单链表

利用 Struct 可以包容多种数据类型的特性,使用它作为链表的节点是最合适不过了。一个结构体内可以包含若干成员,这些成员可以是基本类型、自定义类型、数组类型,也可以是指针类型。这里可以使用指针类型成员来存放下一个节点的地址。

例如,可以定义这样一个结构体类型:

```
type Node struct {  
    Data int  
    Next * Node  
}
```

其中成员 Data 用来存放节点中的有用数据,Next 是指针类型的成员,它指向 Node struct 类型数据,这其实就是下一个节点的数据类型,它和上一个节点应该是同一种类型。

通过 8.7.1 节的介绍应该知道,对于一个链表,链表头部 Head 最重要。所以要建立一个链表,首先要声明一个指向 Node 的指针类型的 Head,然后再让 Head 的 Next 指向链表的第一个节点,如果还有第二个节点,第一个节点的 Next 再指向第二个节点……直到最后一个节点,让它的 Next 为 nil,即链表到此结束。使用这种方法,就可以建立如图 8-2 所示的单向链表。

下面是一个链表综合操作的例子,该例中使用链表动态存储学生基本信息。学生基本信息使用结构体 Student 来记录,共有两个字段:学号(Id)和姓名(Name)。链表节点 Node 也包含两个字段:匿名字段(Student)和 Next 指针。

该例不但要使用 Node 节点建立一个能管理学生信息的单向链表,还要使用 Go 语言面向对象程序设计思想,在 Node 对象上实现 4 个操作方法,以方便对链表的管理和维护。

(1) Creat()方法创建一个新链表,并返回 head 指针。

(2) PrintLink()用于链表的输出打印。

(3) Insert()方法可以将新节点插入链表,并返回 head 指针。

(4) Delete()方法会将节点从链表删除,并返回 head 指针。

在设计过程中,Student 和 Node 的定义,以及 4 个方法的实现在 link 包中完成(具体方法可参见 8.5.2 节),最后程序总体功能的验证和测试在 main 包中完成。由于篇幅所限,4 个操作方法的相关算法实现本教程就不再赘述,有兴趣的读者可以参阅数据结构方面的教程。

Link 包中的代码如下:

```
package link

import (
    "fmt"
)

type Student struct {
    Id int
    Name string
}

type Node struct {
    Student
    Next * Node
}

func (head * Node) Creat() * Node {
    head = nil
    return head
}

func (p * Node) PrintLink().{
    for p != nil {
        fmt.Printf(" %d, %s\n", p.Id, p.Name)
        p = p.Next
    }
}

func (newNode * Node) Insert(head * Node) * Node {
    var p0, p1, p2 * Node
    p0 = newNode
    p1 = head
    if head == nil {
        head = p0
        p0.Next = nil
    } else {
        for (p0.Id > p1.Id) && p1.Next != nil {
            p2 = p1
            p1 = p1.Next
        }
        if p0.Id <= p1.Id {
            if head == p1 {
                head = p0
            } else {
                p2.Next = p0
                p0.Next = p1
            }
        } else {
            p1.Next = p0
            p0.Next = nil
        }
    }
}
```



```

    }
    return head
}

func (delNode * Node) Delete(head * Node) * Node {
    var p1, p2 * Node
    if head == nil {
        fmt.Println("List nil!")
        goto End
    }
    p1 = head
    for delNode.Id != p1.Student.Id && p1.Next != nil {
        p2 = p1
        p1 = p1.Next
    }
    if delNode.Id == p1.Student.Id {
        if p1 == head {
            head = p1.Next
        } else {
            p2.Next = p1.Next
        }
        fmt.Printf("Delete %d\n", delNode.Id)
    } else {
        fmt.Printf("Node %d not been found!\n", delNode.Id)
    }
}
End:
    return head
}

```

例 8-20 链表综合操作。

```

1 //链表综合操作
2 package main
3
4 import(
5     "link"
6 )
7
8 func main() {
9     var head * link.Node
10    stu1 := link.Node{link.Student{100, "李明"}, nil}
11    stu2 := link.Node{link.Student{101, "张晓"}, nil}
12    stu3 := link.Node{link.Student{102, "赵琼"}, nil}
13    stu4 := link.Node{link.Student{103, "王乐"}, nil}
14    //创建新链表
15    head = head.Creat()
16    //插入节点
17    head = stu1.Insert(head)
18    head = stu2.Insert(head)

```



```
19 head = stu3.Insert(head)
20 head = stu4.Insert(head)
21 //输出链表
22 head.PrintLink()
23 //删除节点
24 head = stu3.Delete(head)
25 head.PrintLink()
26 }
```

编译并运行该程序,输出结果为:

```
100,李明
101,张晓
102,赵琼
103,王乐
Delete 102
100,李明
101,张晓
103,王乐
```

通过例 8-20 的测试与验证,基本能体会到 Go 语言面向对象程序设计思想的强大,它没有其他面向对象程序设计语言那么复杂,但设计出来的程序结构清晰而优雅。

小结

本章主要介绍了 Go 语言的结构体和方法,在 Go 语言中没有类与继承的概念,Go 是使用结构体和方法实现面向对象的。Go 语言的结构体和其他语言有所区别,它支持匿名字段和嵌入式结构。另外,对任意一个结构体都可以定义方法,方法的定义和函数类似,只不过限定了函数的作用对象必须是某个结构体。最后通过一些具体事例,详细介绍了 bufio 包中的 ReadWriter、Reader 和 Writer 对象,以及它们的操作方法。

通过这一章的学习,首先要掌握结构体的定义方法,然后就是结构体对象方法的定义方法。这一章的内容比较繁杂,读者可以仔细阅读本章最后的“链表操作”应用举例,好好体会 Go 语言面向对象程序设计思想。

习题

8.1 班上有 30 个学生,每个学生的信息包括学号(num)、姓名(name)、性别(sex)、年龄(age)、三门课的成绩(score[3])。要求建立学生信息的结构体 student,输入这 30 个学生的信息,然后打印输出各项数据。

8.2 有 4 名学生,每个学生包括学号、姓名、成绩,编写函数找出成绩最高学生的学号、姓名和成绩。

8.3 有 4 名学生,每个学生包括学号、姓名、成绩,编写方法找出成绩最高学生的学号、

姓名和成绩。

8.4 有一批图书,每本书有书名(name)、作者(author)、书号(isdn)、出版日期(date) 4项数据,希望既可以通过书名查询,也可以使用作者或书号来查询图书。编写方法来实现此功能,如果查到,打印出此书的书名、作者、书号和出版日期信息。如果查不到此书,则打印出“无此书”。

8.5 有两个单链表 a,b。设节点中包含学号、姓名。从链表 a 中删除所有与链表 b 中“学号”相同的节点,可参考例 8-20。

第9章

接口

Go 语言并不是一种传统意义上的面向对象语言，因为它并没有类和继承这些概念。Go 语言使用 Struct、Method 和 Interface 实现了面向对象的功能，Struct 和 Method 在第 8 章已做了详细的讲解，这一章主要介绍 Interface 的概念、定义和操作。

9.1 接口的概念与定义

Go 语言最重要的特性之一就是提供了接口 (Interface)，它让面向对象，内容组织的实现非常方便。同时 Go 语言中的接口，并不同于其他面向对象程序设计语言 (C++、Java、C# 等) 中接口的概念。

9.1.1 接口的概念

简单地说，Interface 是一组 Method 的组合，可以通过 Interface 来定义对象的一组行为。如果某个对象实现了某个接口的所有方法，就表示它实现了该“接口”，无须显式地在该类型上添加接口说明。

假设定义了两个对象 Teacher 和 Student，Teacher 实现了方法 Reading、Writing 和 Teaching，Student 实现了方法 Reading、Writing 和 Studying。被对象 Teacher 和 Student 实现的方法组合就称为 Interface。

例如，Teacher 和 Student 都实现了 Interface: Reading 和 Writing，也就是说这两个对象是该 Interface 类型。而对象 Student 没有实现 Interface: Reading、Writing 和 Teaching，因为它没实现 Teaching 这个方法。

9.1.2 接口的定义

Interface 的定义形式看上去和 Struct 类似，但是 Interface 是一个方法的集合，它里面没有其他类型变量，而且 Method 只用定义原型不用实现。

Interface 要使用关键字“type”和“interface”定义，基本格式如下：

```
type interfaceName interface{
    methodName(参数列表)(返回值)
    methodName(参数列表)(返回值)
    :
}
```


注意:

(1) 在 Go 语言中,Interface 命名时习惯性以“er”结尾,比如:Printer、Reader、Writer 等,即通常以动名词来命名。其他面向对象语言 Interface 命名时通常以大写字母“I”开头,比如.NET、Java 等。

(2) 在 Go 语言中,一个 Interface 中包含的 Method 不宜过多,一般 0~3 个即可。

(3) 一个 Interface 可以被任意的对象实现;相同地,一个对象也可以实现多个 Interface。

例如:

```
type People struct{
    Name string
}
type Student struct{
    People
    School string
}
type Teacher struct{
    People
    Department string
}
func (p People) SayHi(){}
func (s Student) SayHi(){}
func (t Teacher) SayHi(){}
func (s Student) Study(){}
type Speaker interface{
    SayHi()
}
type Learner interface{
    SayHi()
    Study()
}
```

通过上面的代码可以看出,Speaker 接口被对象 People、Teacher 和 Student 实现。而 Student 对象同时实现了 Speaker 和 Learner 两个接口。

9.1.3 接口组合

在 Go 语言中,除了类型可以匿名组合,接口也可以组合在一起。将一个接口匿名嵌入到另外一个接口中,就是接口组合 (Interface-combination),这种组合类似于接口的继承 (Interface-inheritance)。

例如:

```
type SpeakLearner interface{
    Speaker
    Learner
}
```

该例中接口 `SpeakLearner` 组合了 `Speaker` 和 `Learner` 两个接口,它既能做 `Speaker` 接口的所有事情,又能做 `Learner` 接口的所有事情。

可以认为接口组合是类型匿名组合的一个特殊事例,只不过接口只包含方法,而不包含任何成员变量。

9.1.4 空接口

在 Go 语言中,任何数据类型都默认实现了空 `Interface`,空接口可以这样定义:

```
interface{}
```

空接口也就是包含 0 个 `Method` 的 `Interface`,所以可以使用空接口 `interface{}` 定义任意类型变参函数(见 7.4 节)。一个函数把 `interface{}` 作为参数,那么它可以接受任意类型的值作为参数,如果一个函数返回 `interface{}`,就可以返回任意类型的值。

例如:

```
//函数 test1 返回 1 个 interface{}  
func test1(a interface{}) {}  
//函数 test2 可返回多个 interface{}  
func test2(a ... interface{}) {}
```

Go 语言中的空接口的作用类似于 C 语言中的 `void *`,或者 Java / C# 中的 `System. Object`。

9.2 接口执行机制和赋值

在 Go 语言中,接口作为一种引用数据类型,是可被实例化的类型,当定义了一个接口类型变量时,系统将会为其分配内存,并将赋值给它的对象复制存储到该内存区。

9.2.1 接口执行机制

接口对象内部由两部分组成: `Itab` 指针和 `data` 指针。

```
struct Iface  
{  
    Itab * tab;  
    void * data;  
};
```

`Itab` 依据 `data` 类型创建,存储了接口动态调用的元数据信息,其中包括 `data` 类型所有符合接口签名的方法地址列表。使用接口对象调用方法时,就从 `Itab` 中查找所对应的方法,并将 `* data`(指针)作为 `Receiver` 参数传递给该方法,从而完成实际目标方法调用。

而编译器在构建 `Itab` 时,会区分 `T` 和 `* T` 方法集(`Method sets`),并从中获取接口实现方法的地址指针。接口调用不会做 `Receiver` 自动转换,目标方法必须在接口实现的方法集

中。接口方法集规则如下：

(1) T 仅拥有属于 T 类型的方法集,而 *T 则同时拥有(T+*T)方法集。

(2) 基于 T 实现方法,表示同时实现了 interface(T)和 interface(*T)接口。

(3) 基于 *T 实现方法,那就只能是对 interface(*T)实现接口。

9.2.2 接口的赋值

与其他面向对象语言不同,Go 语言中的接口还可以赋值,如果定义了一个 Interface 的变量,那么这个变量里面可以存储实现这个 Interface 的任意类型的对象。

例如上面例子中,如果定义了一个 Speaker 接口类型的变量 is,那么 is 里面就可以存储对象 People、Teacher 或者 Student 的值。

因为接口类型 is 能够同时持有这三种类型的对象,所以可以定义一个包含 Speaker 接口类型元素的 Slice,这个 Slice 可以被赋予实现了 Speaker 接口的任意结构的对象,这个和传统意义上的 Slice 有所不同。

例 9-1 接口的定义与赋值。

```

1 //接口的定义与赋值
2 package main
3
4 import(
5     "fmt"
6 )
7 //定义对象 People、Teacher 和 Student
8 type People struct {
9     Name string
10 }
11 type Teacher struct {
12     People
13     Department string
14 }
15 type Student struct {
16     People
17     School string
18 }
19 //对象方法实现
20 func (p People) SayHi() {
21     fmt.Printf("Hi,I'm %s. Nice to meet you!\n",p.Name)
22 }
23 func (t Teacher) SayHi() {
24     fmt.Printf("Hi,my name is %s. I'm working in %s.\n",t.Name,t.Department)
25 }
26 func (s Student) SayHi() {
27     fmt.Printf("Hi,my name is %s. I'm studing in %s.\n",s.Name,s.School)
28 }
29 func (s Student) Study() {
30     fmt.Printf("I'm learning Golang in %s.\n",s.School)
31 }

```



```

32 //定义接口 Speaker 和 Learner
33 type Speaker interface {
34     SayHi()
35 }
36 type Learner interface {
37     SayHi()
38     Study()
39 }
40 func main() {
41     people := People{"张三"}
42     teacher := Teacher{People{"郑智"}, "Computer Science"}
43     student := Student{People{"李明"}, "Yale University"}
44     var is Speaker      //定义 Speaker 类型的变量 is
45     is = people         //is 能存储 People
46     is.SayHi()
47     is = teacher       //is 能存储 Teacher
48     is.SayHi()
49     is = student       //is 能存储 Student
50     is.SayHi()
51     var il Learner      //定义 Learner 类型的变量 il
52     il = student       //il 能存储 Student
53     il.Study()
54 }

```

编译并运行该程序,输出结果为:

```

Hi, I'm 张三. Nice to meet you!
Hi, my name is 郑智. I'm working in Computer Science.
Hi, my name is 李明. I'm studing in Yale University.
I'm learning Golang in Yale University.

```

因为接口类型 `is` 能够同时持有这三种类型的对象,所以可以定义一个包含 `Speaker` 接口类型元素的 `Slice`,这个 `Slice` 可以被赋予实现了 `Speaker` 接口的任意结构的对象,这个和传统意义上的 `Slice` 有所不同。

例如,在例 9-1 中添加如下代码,查看运行结果。

```

ix := make([]Speaker, 3)
ix[0], ix[1], ix[2] = people, teacher, student
for _, value := range ix {
    value.SayHi()
}

```

9.3 匿名字段方法和接口转换

当接口的类型对象是 `Struct` 时,这些 `Struct` 也可以嵌入匿名字段,而为这些匿名字段定义的方法也会被接口所“继承”。同时,接口之间还可以相互包含。包含其他接口的方法

签名的集合被称为超集(Superset),而被包含的方法签名的集合被称为子集(Subset)。

超集的定义为:如果一个集合 S2 中的每一个元素都在集合 S1 中,且集合 S1 中可能包含 S2 中没有的元素,则集合 S1 就是 S2 的一个超集。S1 是 S2 的超集,则 S2 是 S1 的真子集,反之亦然。子集的定义为:对于两个非空集合 S1 与 S2,如果集合 S1 的任何一个元素都是集合 S2 的元素,就说 $S1 \subseteq S2$ (读作 S1 包含于 S2),称集合 S1 是集合 S2 的子集。超集的接口往往能被转换成子集的接口,这样子集就可以使用自己的方法访问超集对象实例。

9.3.1 匿名字段方法

在 Go 语言中,接口同样支持由 Struct 匿名字段实现的方法,因为外层结构“继承”了匿名字段的方法集。匿名字段方法集规则如下:

- (1) 如果 S 嵌入匿名类型 T,则 S 方法集包含 T 方法集。
- (2) 如果 S 嵌入匿名类型 *T,则 S 方法集包含 *T 的方法集($T + *T$)。
- (3) 如果 S 嵌入匿名类型 T 或 *T,则 *S 方法集包含 *T 的方法集($T + *T$)。

例 9-2 匿名字段方法。

```

1 //匿名字段方法
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type People struct {
9     Name string
10 }
11 type Teacher struct {
12     People
13     Department string
14 }
15 func (p People) SayHi().{
16     fmt.Printf("Hi,I'm %s. Nice to meet you!\n",p.Name)
17 }
18 type Speaker interface {
19     SayHi()
20 }
21 func main() {
22     people := People{"张三"}
23     teacher := Teacher{People{"郑智"},"Computer Science"}
24     var is Speaker
25     is = &people //为 *People 定义了 SayHi(),自然实现该接口
26     is.SayHi()
27     is = &teacher //匿名字段同样也拥有 SayHi()
28     is.SayHi()
29 }
```


编译并运行该程序,输出结果为:

```
Hi,I'm 张三. Nice to meet you!
Hi,I'm 郑智. Nice to meet you!
```

在例 9-2 中,* People 实现了 Speaker 接口,而 Teacher 嵌入了 People 匿名字段,所以按照匿名字段方法集规则,* Teacher 实现了 Speaker 接口。

9.3.2 接口转换

拥有超集的接口还可以被转换成子集的接口,这样子集就可以很容易地访问超集对象中的成员变量或者数据。

例 9-3 接口的转换。

```
1 //接口的转换
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type People struct {
9     Name string
10    Age int
11 }
12 type Student struct {
13     People
14     School string
15 }
16 type PeopleInfo interface {
17     GetPeopleInfo()
18 }
19 type StudentInfo interface {
20     GetPeopleInfo()
21     GetStudentInfo()
22 }
23 func (p People) GetPeopleInfo() {
24     fmt.Println(p)
25 }
26 func (s Student) GetStudentInfo() {
27     fmt.Println(s)
28 }
29 func main() {
30     var is StudentInfo = Student{People{"李明",18},"Yale University"}
31     is.GetStudentInfo()
32     is.GetPeopleInfo()
33     var ip PeopleInfo = is
34     ip.GetPeopleInfo()
35 }
```


编译并运行该程序,输出结果为:

```
{{李明 18} Yale University}
{李明 18}
{李明 18}
```

在例 9-3 中,接口 `StudentInfo` 拥有 `PeopleInfo` 的全部方法签名,也就是说 `PeopleInfo` 属于 `StudentInfo` 的一个子集,所以可以直接将接口类型变量 `is` 赋值给 `ip`,`ip` 就可以使用方法 `GetPeopleInfo()` 直接访问超集中的对象 `People` 了。

9.4 接口类型推断

通过接口的赋值可知,接口类型变量里面可以存储任意类型的数值(该类型实现了 `Interface`)。那么如何反向知道接口类型变量里面实际保存的是哪一种类型的对象呢?这就是接口类型推断。

利用接口类型推断,可以判断接口对象是否是某个具体的接口或者类型。在 Go 语言中,目前有两种常用的方法可以进行接口类型推断: `Comma-ok` 断言和 `Switch` 测试。

9.4.1 Comma-ok 断言

使用 `Comma-ok` 断言可以进行接口类型查询,格式如下:

```
value, ok = element.(T)
```

上式中 `value` 是变量的值,`ok` 是 `bool` 类型,`element` 是接口类型变量,`T` 是断言的类型。如果 `element` 里面确实存储了 `T` 类型的数值,那么 `ok` 返回 `true`,否则返回 `false`。

例 9-4 利用 `Comma-ok` 断言进行接口类型推断。

```
1 //利用 Comma-ok 断言进行接口类型推断
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type People struct {
9     Name string
10    Age int
11 }
12 //定义空接口用于存储任意数据类型
13 type Tester interface{}
14
15 func main() {
16     people := People{"张三",20}
17     it := make([]Tester,4)
```

```

18  it[0], it[1], it[2], it[3] = 1, "Hello", people, true
19  for index, element := range it {
20      if value, ok := element.(int); ok {
21          fmt.Printf("it[ %d] is an int. value = %d\n", index, value)
22      } else if value, ok := element.(string); ok {
23          fmt.Printf("it[ %d] is a string. value = %s\n", index, value)
24      } else if value, ok := element.(People); ok {
25          fmt.Printf("it[ %d] is a People. value = %v\n", index, value)
26      } else {
27          fmt.Printf("it[ %d] is an unknown type.\n", index)
28      }
29  }
30 }

```

编译并运行该程序,输出结果为:

```

it[0] is an int. value = 1
it[1] is a string. value = Hello
it[2] is a People. value = {张三 20}
it[3] is an unknown type.

```

9.4.2 Switch 测试

除了使用 Comma-ok 断言,还可以使用 Switch 测试推断接口类型,而且程序结构更简洁,格式如下:

```

switch value := element.(type){
    case int:
    case string:
    :
}

```

例 9-5 如果使用 Switch 测试进行接口类型推断,则代码如下。

```

1 //利用 Switch 测试进行接口类型推断
2 package main
3
4 import(
5     "fmt"
6 )
7
8 type People struct {
9     Name string
10    Age int
11 }
12 //定义空接口用于存储任意数据类型
13 type Tester interface{}

```

```
14
15 func main() {
16     people := People{"张三", 20}
17     it := make([]Tester, 4)
18     it[0], it[1], it[2], it[3] = 1, "Hello", people, true
19     for index, element := range it {
20         switch value := element.(type) {
21             case int:
22                 fmt.Printf("it[ %d] is an int. value = %d\n", index, value)
23             case string:
24                 fmt.Printf("it[ %d] is a string. value = %s\n", index, value)
25             case People:
26                 fmt.Printf("it[ %d] is a People. value = %v\n", index, value)
27             default:
28                 fmt.Printf("it[ %d] is an unknown type.\n", index)
29         }
30     }
31 }
```

编译并运行该程序,输出结果为:

```
it[0] is an int. value = 1
it[1] is a string. value = Hello
it[2] is a People. value = {张三 20}
it[3] is an unknown type.
```

需要强调的是: `element.(type)` 语法不能在 Switch 语句外的任何逻辑语句里面使用,如果要在 Switch 外面判断一个类型就要使用 Comma-ok 断言。

9.5 反射

反射(Reflection)是 Go 语言获取程序运行时类型信息的方式,程序员可以利用这些类型信息进行一些更加灵活的处理工作。Go 语言标准库的 `reflect` 包中提供了 `Typeof()` 和 `Valueof()` 函数,可以使用这两个函数从 `interface{}` 接口对象中获取实际目标对象的类型和值信息。

反射可以大大提高程序开发的灵活性,借助于反射可以让静态语言具备更加多样的运行时动态特征。反射还可以让程序具备自省能力,使得 `interface{}` 接口对象的灵活性有更大的发挥余地。本节主要介绍反射在 Go 语言中的具体实现,以及反射的基本使用方法。

9.5.1 获取原对象的 Type 和 Value 值

前面讲到,利用接口反射可以获取对象的 Type 和 Value 值。Type 和 Value 是 Go 语言 `reflect` 包中最重要的两个类型,对所有接口进行反射操作,都可以得到一个包含 Type 和 Value 的信息结构。Type 是被反射对象的类型信息,Value 是该对象的值信息。可以使用 `reflect` 包中 `Typeof()` 函数获取被反射对象的类型信息,使用 `Valueof()` 函数获取被反射对

象的值信息。Typeof() 和 Valueof() 函数的原型定义如下：

```
func TypeOf(i interface{}) Type
func ValueOf(i interface{}) Value
```

在操作时,变量类型信息是通过 interface{} 接口传递给 Typeof() 函数的,如果调用 Typeof(nil) 将返回“nil”;变量值信息也是通过 interface{} 接口传递给 ValueOf() 函数的,如果调用 Typeof(nil) 将返回“0”。

在 reflect 包中还提供了 Type 和 Value 大量的方法,Type 的一个非常有用的方法是 kind(), 这个方法可以返回该类型的具体信息,比如 int、int8、float64 等。Value 则包含一系列同类型相关的方法,比如 int()、float()、bool() 等,用于返回对应类型的值。例如:

```
package main
import (
    "fmt"
    "reflect"
)
func main() {
    var pi float64 = 3.14
    t := reflect.TypeOf(pi)
    v := reflect.ValueOf(pi)
    fmt.Println("Type:", t.Kind())
    fmt.Println("Value", v.Float())
}
```

该例测试结果为:

```
Type: float64
Value 3.14
```

Go 语言除了能对最基本的数据类型进行反射操作,还能够对结构体进行反射操作。反射操作可以获取结构体各字段的类型信息和值信息,也可以获取结构体对象的方法信息。在例 9-6 中,演示了如何获取结构体的字段信息和方法信息。

例 9-6 利用接口反射获取结构体 Type、Value 和 Method。

```
1 //利用接口反射获取结构体 Type、Value 和 Method
2 package main
3 //导入 reflect 包
4 import(
5     "fmt"
6     "reflect"
7 )
8 //定义结构体
9 type Student struct {
10     Id      int
11     Name    string
12     Sex     bool
```

```
13     Grade float32
14 }
15 //为结构体定义两个方法
16 func (s Student) SayHi() {
17     fmt.Printf("Hi,nice to meet you!\n")
18 }
19 func (s Student) MyName() {
20     fmt.Printf("My name is %s,I come from China.",s.Name)
21 }
22 //反射处理函数
23 func StructInfo(o interface{}) {
24     t := reflect.TypeOf(o)
25     //判断是否为结构体类型
26     if k := t.Kind(); k != reflect.Struct {
27         fmt.Printf("This value is not a struct,it's %v.",k)
28         return
29     }
30     fmt.Println("Struct name is:",t.Name())
31     fmt.Println("Fields of the struct is:")
32     v := reflect.ValueOf(o)
33     //获取字段 Type 和 Value 信息
34     for i := 0; i < t.NumField(); i++{
35         field := t.Field(i)
36         value := v.Field(i).Interface()
37         fmt.Printf(" %6s: %v = %v\n",field.Name,field.Type,value)
38     }
39     fmt.Println("Methods of the struct is:")
40     //获取方法 Name 和 Type 信息
41     for i := 0; i < t.NumMethod(); i++{
42         method := t.Method(i)
43         fmt.Printf(" %6s: %v\n",method.Name,method.Type)
44     }
45 func main() {
46     stu := Student{10001,"李明",false,90.5}
47     StructInfo(stu)
48 }
```

编译并运行该程序,输出结果为:

```
Struct name is: Student
Fields of the struct is:
    Id:int = 10001
    Name:string = 李明
    Sex:bool = false
    Grade:float32 = 90.5
Methods of the struct is:
    MyName:func(main.Student)
    SayHi:func(main.Student)
```


在例 9-6 中,当定义结构体字段名和方法名时要注意,这些名字的首写字母一定要大写,否则反射操作时编译器会报错。

另外,反射会将匿名字段当作一个独立字段来处理,如果要获取嵌入字段的 Type 和 Value 信息,必须使用索引路径来完成。通过 Value 类型的 FieldByIndex() 方法可以获取嵌入字段的索引路径,FieldByIndex() 方法的原型定义如下:

```
func (v Value) FieldByIndex(index []int) Value
```

FieldByIndex() 方法成功执行会返回嵌入字段的索引路径,如果对一个非 Struct 类型执行 FieldByIndex() 方法则会产生 panic 错误事件。

例如如下代码:

```
type Student struct {
    Id      int
    Name    string
    Sex     bool
    Grade   float32
}
type Monitor struct {
    Student
    As string
}
func main() {
    stu := Monitor{Student{10001, "李明", false, 90.5}, "班长"}
    t := reflect.TypeOf(stu)
    v := reflect.ValueOf(stu)
    for i := 0; i < t.NumField(); i++ {
        if t.Field(i).Anonymous {
            for j := 0; j < v.Field(i).NumField(); j++ {
                fmt.Println("Embedded fields:", v.FieldByIndex([]int{i, j}).Interface())
            }
        } else {
            fmt.Println("Normal fields:", v.Field(i).Interface())
        }
    }
}
```

该例测试结果为:

```
Embedded fields: 10001
Embedded fields: 李明
Embedded fields: false
Embedded fields: 90.5
Normal fields 班长
```

该程序首先使用 Anonymous() 方法依次判断每个字段是否为匿名字段,如果是则使用 FieldByIndex() 方法索引访问每一个嵌入式字段的值,否则正常访问这些字段的值。

9.5.2 修改原对象 Value 值

前面讲的都是如何利用反射提取一个数据对象的 Type 和 Value 信息,另外,利用反射还可以修改原数据对象的 Value 值。在 Go 语言中,要利用反射修改原数据对象的前提是该对象是“可修改属性(Settability)”的。可以使用 CanSet()方法判断一个对象是否可修改属性,CanSet()方法的原型定义如下:

```
func (v Value) CanSet() bool
```

如果一个对象的 Value 值可修改,CanSet()方法返回“true”,进而就可以调用 Set()方法或 SetXxx()方法修改对象的 Value 值(例如 SetInt()、SetBool()、SetFloat()等);如果对象的 Value 值不可修改,则 CanSet()方法返回“false”,并产生 panic 错误事件。

还有一点需注意的是,在 Go 语言中所有类型是值类型,即这些变量在传递给函数时将发生一次复制,所以在函数中只能对变量的副本进行修改。要想在函数中对变量本身进行修改,必须使用引用类型,即指针。所以要使用反射修改原对象的值,在使用 interface{}接口进行传递时,这个 interface{}必须是指针类型的。另外,要想操作目标对象,还要使用 Elem()方法进一步获取指针指向的实际目标。

例 9-7 利用反射修改原对象 Value 值。

```
1 //利用反射修改原对象 Value 值
2 package main
3
4 import(
5     "fmt"
6     "reflect"
7 )
8
9 type Student struct {
10     Id int
11     Name string
12     Sex bool
13     Grade float32
14 }
15 func SetValue(o interface{}) {
16     v := reflect.ValueOf(o)
17     if v.Kind() != reflect.Ptr || !v.Elem().CanSet() {
18         fmt.Println("Cannot set!")
19         return
20     } else {
21         v = v.Elem()
22     }
23     for i := 0; i < v.NumField(); i++{
24         switch v.Field(i).Kind() {
25             case reflect.Int:
26                 v.Field(i).SetInt(10002)
```

```
27     case reflect.String:
28         v.Field(i).SetString("张衡")
29     case reflect.Bool:
30         v.Field(i).SetBool(true)
31     case reflect.Float32:
32         v.Field(i).SetFloat(95.5)
33     }
34 }
35 }
36 func main() {
37     stu := Student{10001, "李明", false, 90.5}
38     SetValue(&stu)
39     fmt.Println(stu)
40 }
```

编译并运行该程序,输出结果为:

```
{10002 张衡 true 95.5}
```

在例 9-7 中,首先判断被 `interface{}` 接口传递过来的对象是否是指针类型(Ptr),如果是,则调用 `Elem()` 方法获取指向实际对象的指针,然后再判断该对象是否是“Settability”的,如果是,则调用相应的 `SetXxx()` 方法修改对象的值。否则函数 `SetValue()` 返回,原对象的 Value 值不会被修改。

9.5.3 动态调用原对象方法

前面讲了利用反射可以提取对象的 Type 和 Value 值,另外还可以利用反射“动态”调用原对象的方法。在 `reflect` 包中提供了 `Call()` 方法可以用于调用原对象的方法。`Call()` 的原型定义如下:

```
func (v Value) Call(in []Value) []Value
```

`Call()` 方法可以向原对象传递参数列表并调用它,例如:

```
if len(in) == 3{
    v.Call(in)
}
```

该例中 `Call()` 调用了原对象 `v` 的方法,并向其传递了三个参数: `in[0]`、`in[1]` 和 `in[2]`。如果调用成功,将返回正常输出结果;如果调用的是一个非方法类型,则 `Call()` 会引发 panic 错误事件。

一个原对象通常会有多个方法,那怎样才能调用到指定的方法呢? 可以使用 `reflect` 包中的 `MethodByName()` 方法来获取想要的方法的 Value 值, `MethodByName()` 的原型定义如下:

```
func (v Value) MethodByName(name string) Value
```

MethodByName()方法使用原对象的方法名(name)获取该方法的 Value 值,如果所访问的方法不存在,MethodByName()会返回“0”。

最后还需注意的是,在 Go 语言中传递给方法的参数要和方法定义的参数类型保持一致。另外,为了处理变参这种复杂情况,传递给被调用方法的参数通常首先被保存在一个 Slice 中,然后再复制到参数列表中。

例 9-8 利用反射动态调用原对象方法。

```
1 //利用反射动态调用原对象方法
2 package main
3
4 import(
5     "fmt"
6     "reflect"
7 )
8
9 type Student struct {
10     Id int
11     Name string
12     Sex bool
13     Grade float32
14 }
15 func (s Student) SayHi(name string) {
16     fmt.Printf("Hi %s, my name is %s.\n", name, s.Name)
17 }
18 func main() {
19     stu := Student{10001, "李明", false, 90.5}
20     v := reflect.ValueOf(stu)
21     mv := v.MethodByName("SayHi")
22     args := []reflect.Value{reflect.ValueOf("张衡")}
23     mv.Call(args)
24 }
```

编译并运行该程序,输出结果为:

```
Hi 张衡, my name is 李明.
```

9.6 应用举例——二叉树

树型结构(Tree)是一种重要的非线性数据结构,它为计算机应用中出现的具有层次关系的数据提供了一种有效的表示方法,比如文件目录结构、源程序语法结构等。

9.6.1 树的定义和基本术语

树是 $n(n \geq 0)$ 个节点的有限集合 T 。在任意一棵非空树中满足如下两个条件：

- (1) 有且仅有一个根节点(Root)。
- (2) 当 $n > 1$ 时,其余节点可分为 $m(m \geq 0)$ 个互不相交的有限集合 T_1, T_2, \dots, T_m , 其中每一个集合本身又都是一棵树, 并且称为根的子树(Subtree), 如图 9-1 所示。

由上可知,树的定义是递归的,树是一种递归数据结构。树的这种定义为树的递归处理带来了很大方便,本节举例中几乎所有对树的处理都采用了递归算法。

在了解树型结构时,还有几个基本概念非常重要,必须要掌握。

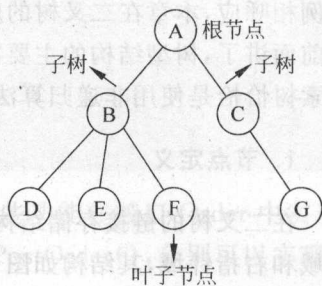


图 9-1 树型结构

- (1) 节点的度：树中每个节点具有的子树数,或后继节点数称为该节点的度。
- (2) 树的度：树中所有节点的度的最大值称为树的度。
- (3) 分支节点：度大于 0 的节点称为分支节点或非终端节点。
- (4) 叶子节点：度为 0 的节点称为叶子节点或终端节点。
- (5) 儿子节点：一个节点的后继称为该节点的儿子节点。
- (6) 父亲节点：一个节点称为其后继节点的父亲节点。
- (7) 子孙节点：一个节点的所有子树中的节点称为该节点的子孙节点。
- (8) 祖先节点：从根节点到达一个节点的路径上,通过的所有节点称为该节点的祖先节点。
- (9) 兄弟节点：具有同一父亲的节点相互称为兄弟节点。
- (10) 节点的层数：树是一种层次结构,根节点为第一层,其儿子节点为第二层,以此类推可以得到每个节点的层数。
- (11) 树的深度：树中节点的最大层数称为树的深度或高度。
- (12) 森林：0 个或多个不相交的树的集合称为森林。

9.6.2 二叉树简介

二叉树(Binary Tree)是一种特殊的树型结构,二叉树中每个节点至多有两棵子树,称为左子树、右子树。即二叉树中不存在度大于 2 的节点,且两棵子树有左右之分,次序不能任意颠倒。

除了这些基本特征外,还有如下一些特殊的二叉树。

(1) 满二叉树：在一棵二叉树中,若第 i 层的节点数为 2^{i-1} 时,则称此层的节点数是满的,当二叉树中的每一层都是满的,则称此二叉树为满二叉树。

(2) 完全二叉树：在一棵二叉树中,除最后一层外,若其余各层都是满的,并且最后一层或者是满的,或者是在右边缺少连续若干个节点,则此二叉树被称为完全二叉树。

9.6.3 二叉树的链接存储结构

二叉树的存储方法有顺序存储、链接存储和线索树存储等几种存储方法,顺序存储是使用数组来完成,而链接存储和线索树存储都使用了链表来完成。为了和第8章链表的应用举例相呼应,本章在二叉树的应用举例中使用了链接存储法。为什么不使用线索树呢?原因前面讲了,树型结构的主要特征是递归,所以对树的所有操作都使用递归算法来实现,而线索树恰恰是使用非递归算法来加速遍历速度。

1. 节点定义

在二叉树的链接存储结构中,通常采用的方法是:每个节点设置三个域,即值域、左指针域和右指针域,其结构如图9-2所示。

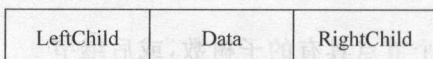


图9-2 二叉树链接存储结构

其中 Data 表示值域,用于存储放入节点中的数据元素,Left 和 Right 分别表示左指针域和右指针域,用以分别存储左孩子和右孩子节点的指针地址。

链接存储的指针类型和节点定义如下:

```
type Node struct {
    Left *Node
    Data interface{}
    Right *Node
}
```

这里的 Data 字段可以是任意基本数据类型,如 int、float、string 等。在第8章链表应用举例中,链表所有节点的 Data 字段类型一致,比如同为 int 型或同为 string 型。在本节的应用举例中,二叉树节点的 Data 字段将被设置成空接口 interface{},这样二叉树的叶子节点将能够存储不同类型的数据,比如一个节点为 int 型,其他节点可以为 string 型等。

2. 接口定义

二叉树的应用处理功能主要包括:二叉树新节点的创建、初始化;二叉树的输出、度的计算、叶子节点统计等基本操作;二叉树的前序、中序、后序遍历等。针对这些功能本例共定义了三个接口:Initer、Operater 和 Order。

(1) Initer 接口。当为二叉树创建了一个新节点时,Initer 接口提供了方法 SetData() 可以对节点的 Data 字段进行初始化。

Initer 接口定义如下:

```
type Initer interface {
    SetData(data interface{})
}
```

(2) Operater 接口。当已经生成了一个二叉树时,可以使用 Operater 接口提供的三个方法: PrintBT()、Depth() 和 LeafCount(),对二叉树进行输出、深度计算和叶子统计等基本操作。

Operater 接口定义如下:

```
type Operater interface {  
    PrintBT()  
    Depth() int  
    LeafCount() int  
}
```

(3) Order 接口。对二叉树的遍历是一个重要的功能,这些功能在接口 Order 中实现,接口 Order 中共定义了三个方法: PreOrder()、InOrder()和 PostOrder(),分别可以实现前序遍历、中序遍历和后序遍历。

Order 接口的定义如下:

```
type Order interface {  
    PreOrder()  
    InOrder()  
    PostOrder()  
}
```

可以看到,在定义接口时,接口中的方法只需声明原型,而并不需在此实现。这些方法的实现可以在其他 Go 包文件中实现,或者直接由其他用户提供。接口的意义也在于此,即同一个接口可以有不同的实现方法,甚至由不同的人去完成。所以,Go 语言是软件工程类的高级程序设计语言。

3. 方法的实现

通过接口的概念知道接口是方法的组合,而在定义接口时不必马上实现方法,方法可由设计者自己或交由其他人另外单独设计。本例三个接口中共有 7 个方法,在此分别介绍它们的实现算法。

(1) SetData 方法。SetData()通过空接口 interface{},可以将任意类型数据赋值给二叉树节点 Node 的 Data 字段,实现二叉树对任意数据类型的存储。

SetData 方法的定义如下:

```
func (n * Node) SetData(data interface{}) {  
    n.Data = data  
}
```

(2) PrintBT 方法。PrintBT()调用底层函数 PrintBT(),输出一个给定二叉树的嵌套括号表示。

PrintBT 方法的定义如下:


```
func (n * Node) PrintBT() {
    PrintBT(n)
}
```

(3) Depth 方法。Depth()调用底层函数 Depth(),返回二叉树的深度。

Depth 方法的定义如下:

```
func (n * Node) Depth() int {
    return Depth(n)
}
```

(4) LeafCount 方法。LeafCount()调用底层函数 LeafCount(),返回二叉树的叶子节点数。

LeafCount 方法的定义如下:

```
func (n * Node) LeafCount() int {
    return LeafCount(n)
}
```

(5) PreOrder 方法。PreOrder()调用底层函数 PreOrder()对二叉树进行前序遍历。

PreOrder 方法的定义如下:

```
func (n * Node) PreOrder() {
    PreOrder(n)
}
```

(6) InOrder 方法。InOrder()调用底层函数 InOrder()对二叉树进行中序遍历。

InOrder 方法的定义如下:

```
func (n * Node) InOrder() {
    InOrder(n)
}
```

(7) PostOrder 方法。PostOrder()调用底层函数 PostOrder()对二叉树进行后序遍历。

PostOrder 方法的定义如下:

```
func (n * Node) PostOrder() {
    PostOrder(n)
}
```

4. 底层函数设计

通过第 8 章的学习可知,在面向对象程序设计中,上层方法的功能实现还要依赖底层函数,本例中大部分方法也是这样,现将所有底层函数一一列举如下。

(1) NewNode 函数。NewNode()按照链接存储方式生成一个新的二叉树节点,参数

left 指向左指针域, 参数 right 指向右指针域。

NewNode 函数的定义如下:

```
func NewNode(left, right * Node) * Node {  
    return &Node{left, nil, right}  
}
```

(2) PrintBT 函数。PrintBT() 用于输出一个给定二叉树的嵌套括号表示, 采用递归算法: 首先输出根节点, 然后再依次输出左子树和右子树, 在输出左子树前打印输出左括号“(”, 在输出右子树后打印输出右括号“)”; 另外, 依次输出的左、右子树要至少有一个不为空, 若都为空就不必输出了。

PrintBT 函数的定义如下:

```
func PrintBT(n * Node) {  
    if n != nil {  
        fmt.Printf("%v ", n.Data)  
        if n.Left != nil || n.Right != nil {  
            fmt.Printf("( ")  
            PrintBT(n.Left)  
            if n.Right != nil {  
                fmt.Printf(", ")  
            }  
            PrintBT(n.Right)  
            fmt.Printf(") ")  
        }  
    }  
}
```

(3) Depth 函数。Depth() 用于计算二叉树的深度, 采用递归算法: 若一棵二叉树为空, 则其深度为 0; 否则, 其深度等于左子树或右子树的最大深度加 1。

Depth 函数的定义如下:

```
func Depth(n * Node) int {  
    var depleft, depright int  
    if n == nil {  
        return 0  
    } else {  
        depleft = Depth(n.Left)  
        depright = Depth(n.Right)  
        if depleft > depright {  
            return depleft + 1  
        } else {  
            return depright + 1  
        }  
    }  
}
```

(4) LeafCount 函数。LeafCount()用于统计二叉树叶子节点数,采用递归算法:若一棵二叉树为空,则其叶子节点数为0;若一棵二叉树的左、右子树均为空,则其叶子节点数为1;否则叶子数等于左子树与右子树叶子总数之和。

LeafCount 函数的定义如下:

```
func LeafCount(n * Node) int {
    if n == nil {
        return 0
    } else if (n.Left == nil) && (n.Right == nil) {
        return 1
    } else {
        return (LeafCount(n.Left) + LeafCount(n.Right))
    }
}
```

(5) PreOrder 函数。PreOrder()可以对二叉树进行前序遍历,采用递归算法:按照先访问根节点,再访问左子树,最后访问右子树的次序访问二叉树中的所有节点,且每个节点仅访问一次。

PreOrder 函数的定义如下:

```
func PreOrder(n * Node) {
    if n != nil {
        fmt.Printf("%v ", n.Data)
        PreOrder(n.Left)
        PreOrder(n.Right)
    }
}
```

(6) InOrder 函数。InOrder()可以对二叉树进行中序遍历,采用递归算法:按照先访问左子树,再访问根节点,最后访问右子树的次序访问二叉树中的所有节点,且每个节点仅访问一次。

InOrder 函数的定义如下:

```
func InOrder(n * Node) {
    if n != nil {
        PreOrder(n.Left)
        fmt.Printf("%v ", n.Data)
        PreOrder(n.Right)
    }
}
```

(7) PostOrder 函数。PostOrder()可以对二叉树进行后序遍历,采用递归算法:按照先访问左子树,再访问右子树,最后访问根节点的次序访问二叉树中的所有节点,且每个节点仅访问一次。

PostOrder 函数的定义如下:


```
func PostOrder(n * Node) {
    PreOrder(n.Left)
    PreOrder(n.Right)
    fmt.Printf("% v ", n.Data)
}
```

9.6.4 二叉树基本应用测试

前面介绍了二叉树链接存储结构,以及节点定义、接口定义、方法实现和底层函数设计,并在包 btree 中实现。本节将利用这些知识实现几个二叉树的基本应用,包括二叉树的创建、基本操作和遍历。

1. 二叉树创建

二叉树的创建过程一般如下:

- (1) 首先调用 NewNode() 函数创建根节点,再调用 SetData() 方法初始化根节点。
- (2) 使用同样的方法创建左子树和右子树,并将左、右子树链接到根节点上。
- (3) 如果左、右子树有叶子节点,则插入叶子节点并和左、右子树建立链接。

例 9-9 二叉树的建立,本例将演示如何使用 Initer 接口实现根节点的初始化。

```
1 //二叉树的建立
2 package main
3
4 import(
5     "fmt"
6     "btree"
7 )
8
9 func main() {
10     //创建根节点
11     root := NewNode(nil, nil)
12     var it Initer
13     it = root
14     it.SetData("root node")
15     //创建左子树
16     a := NewNode(nil, nil)
17     a.SetData("left node")
18     al := NewNode(nil, nil)    //左叶子节点
19     al.SetData(100)
20     ar := NewNode(nil, nil)    //右叶子节点
21     ar.SetData(3.14)
22     a.Left = al
23     a.Right = ar
24     //创建右子树
25     b := NewNode(nil, nil)
26     b.SetData("right node")
```

```

27     root.Left = a
28     root.Right = b
29     root.PrintBT()
30 }

```

编译并运行该程序,输出结果为:

```
root node ( left node ( 100,3.14 ),right node )
```

通过输出结果可以看出,在例 9-9 中首先创建了根节点 root node,然后创建左子树 left node 和右子树 right node。左子树有两个叶子节点,左叶子的值为 int 型“100”,右叶子的值为 float 型“3.14”。即该二叉树可以存储不同类型的值,这些都是由空接口 interface{} 实现的。

2. 二叉树基本操作

对一个已存在的二叉树的基本操作包括二叉树的输出、深度计算、叶子数统计等,在例 9-10 中将演示如何使用 Operator 接口实现这些基本操作。

例 9-10 二叉树的基本操作。

```

1 //二叉树的基本操作
2 package main
3
4 import(
5     "fmt"
6     "btree"
7 )
8
9 func main() {
10     //创建二叉树
11     root := NewNode(nil,nil)
12     root.SetData("root node")
13     a := NewNode(nil,nil)
14     a.SetData("left node")
15     al := NewNode(nil,nil)
16     al.SetData(100)
17     ar := NewNode(nil,nil)
18     ar.SetData(3.14)
19     a.Left = al
20     a.Right = ar
21     b := NewNode(nil,nil)
22     b.SetData("right node")
23     root.Left = a
24     root.Right = b
25     //使用 Operator 接口实现对二叉树的基本操作
26     var it Operator
27     it = root

```

```
28     it.PrintBT()
29     fmt.Println()
30     fmt.Println("The depths of the Btree is:", it.Depth())
31     fmt.Println("The leaf counts of the Btree is:", it.LeafCount())
32 }
```

编译并运行该程序,输出结果为:

```
root node ( left node ( 100,3.14 ),right node )
The depths of the Btree is: 3
The leaf counts of the Btree is: 3
```

通过输出结果可以看出,例 9-10 中二叉树的深度为 3,叶子节点数也为 3,这和验证结果完全一致。另外,对二叉树的操作还有查找、插入、删除节点等,读者可以在上面的基础上自行完成。

3. 二叉树遍历

在二叉树的一些基本应用中,常常需要在树中查找具有某种特征的节点,或者对树中全部节点逐一进行某种处理,这就是二叉树的遍历(Traversing binary tree)。即如何按某条搜索路径访问树中的每个节点,使得每个节点均能被访问一次,而且仅被访问一次。二叉树的遍历方法一般分为前序遍历、中序遍历和后序遍历,例 9-11 将演示如何使用 Order 接口实现二叉树的三种遍历。

例 9-11 二叉树的遍历。

```
1 //二叉树的遍历
2 package main
3
4 import(
5     "fmt"
6     "btree"
7 )
8
9 func main() {
10     //创建二叉树
11     root := NewNode(nil,nil)
12     root.SetData("root node")
13     a := NewNode(nil,nil)
14     a.SetData("left node")
15     al := NewNode(nil,nil)
16     al.SetData(100)
17     ar := NewNode(nil,nil)
18     ar.SetData(3.14)
19     a.Left = al
20     a.Right = ar
21     b := NewNode(nil,nil)
22     b.SetData("right node")
```



```

23     root.Left = a
24     root.Right = b
25     //使用 Order 接口实现对二叉树的遍历
26     var it Order
27     it = root
28     it.PreOrder()           //先序遍历
29     fmt.Println()
30     it.InOrder()           //中序遍历
31     fmt.Println()
32     it.PostOrder()         //后序遍历
33 }

```

编译并运行该程序,输出结果为:

```

root node   left node   100   3.14   right node
left node   100   3.14   root node   right node
left node   100   3.14   right node   root node

```

通过输出结果可以看出,例 9-11 的三种遍历方法输出结果和验证结果完全一致。当然在对二叉树进行遍历的同时,用户也可以对树中的节点做各种处理,比如修改节点信息等。

小结

本章主要介绍了 Go 语言中接口的概念,以及接口的定义、执行机制和类型推断。在 Go 语言中,接口的主要作用是和结构体和方法配合实现面向对象编程。另外,本章还介绍了接口类型推断方法 Comma-ok 断言和 Switch 测试。最后介绍 Go 语言中的反射机制,包括如何利用“反射”动态获取程序运行时的类型信息。

通过这一章的学习,首先要掌握接口的定义方法,特别是 Go 语言中的接口还可以赋值。重点要掌握接口执行机制,熟悉接口转换等功能。本章最后精心准备了“二叉树”这个典型事例,细致入微地诠释了 Go 面向对象程序设计思想。读者应把握好 Go 面向对象程序设计的三个基本步骤:接口定义、方法实现和底层函数设计。

习题

9.1 简单地说,Interface 是一组_____的组合,可以通过 Interface 来定义对象的一组行为。

9.2 参见接口的定义、执行机制、接口的赋值这些知识内容,阅读下面的程序回答问题:

```

type Data struct {
    Num int
}

func (p * Data) Get() int {

```

```
        return p.Num
    }

    func (p *Data) Put(v int) {
        p.Num = v
    }

    type InOuter interface {
        Get() int
        Put(v int)
    }

    func main() {
        d := Data{100}
        var id InOuter
        id = &d
        fmt.Println(id.Get())
        id.Put(10)
        fmt.Println(id.Get())
    }
```

- (1) 你能看出接口 InOuter 的方法 Get() 和 Put() 的基本功能吗?
- (2) 该程序运行后的两个输出结果分别是多少?

9.3 定义一个任意类型键值对的 Map, 初始化并打印该 Map。可参考 6.3 节内容, 使用空接口 interface{} 定义键值对数据类型。

9.4 定义一个容量为 8 的空接口 Slice, 并进行初始化, 然后使用 Comma-ok 断言推断 Slice 中各元素的数据类型。(可参考例 9-4。)

9.5 每个学生的信息包括学号(num)、姓名(name)、性别(sex)、成绩(score), 请定义学生信息结构体对象 student, 并初始化。最后要求使用反射原理修改学生基本信息, 并输出修改后的结果。

第10章

Go 并发程序设计

Go 语言区别于其他语言的一大特色就是支持并发编程模式,Go 语言使用关键字 `go`, 通过 Goroutine 实现了程序的并发设计。使用 Go 语言开发服务器端程序时,就需要对它的并发机制有深入了解。

10.1 程序并发执行概述

在多任务操作系统(Operating System, OS)中,进程和线程的执行具有并发性,并发是指在一段时间内,多个任务可以共享系统资源,同时执行。而并行是指从某个时刻开始,多个任务同时执行。

10.1.1 程序的顺序执行

在编写程序源代码时,都是一条语句一条语句顺序排列的,如果系统中只有一个程序,那么程序执行时也是按照程序语句排列先后次序,一条一条地执行下去,就像工厂生产流水线加工方式那样,这种程序设计方式就叫做顺序程序设计。

在早期单任务、单处理机系统环境中,内存中只有一道程序作业在执行。一个程序作业完成后,下一个程序作业才能进入内存继续执行。这样一来,计算机上程序的执行过程就严格地按顺序方式进行,这种顺序程序执行有三个主要特点。

(1) 程序语句在计算机上严格按顺序执行。每条程序语句的执行都以前一条语句的结束为前提条件。也就是说,除了人为干预造成机器暂时停顿外,前一条语句的结束就意味着后一条语句的开始。这样,程序语句和计算机执行它的活动严格一一对应。

(2) 只有程序本身的动作才能改变程序的运行环境。就是说,一个程序在计算机中运行时独占全部系统资源。因此除了初始状态外,只有程序本身规定的动作才能改变这些资源的状态。

(3) 程序的执行结果与程序的运行速度无关。就是说,处理机在执行程序任何两条语句之间的停顿,对程序的运算结果不发生影响。

顺序程序的上述特点概括起来就是程序的封闭性和可再现性。所谓封闭性就是指程序一旦运行起来,其计算结果仅取决于程序本身,即运行结果唯一。所谓可再现性是指同一程序可反复执行,且每次执行结果相同。

10.1.2 程序的并发执行

程序的顺序执行限制了系统内存中只能有一道程序作业,这显然限制了系统性能发挥,且资源利用率不高。所以,现代操作系统都支持程序的并发执行。所谓并发执行是指在同一时间间隔内,多个程序可以“同时”执行。

在单处理机系统中,进程(或线程)通过时间片或者出让控制权来实现任务切换,以达到“同时”运行多个程序的目的。这就是所谓的程序并发执行,但实际上任何时刻都只有一个任务被执行,其他任务则通过某种算法来排队准备执行。即宏观上多个程序任务是“同时”执行,但微观上各任务还是一个一个地顺序执行。

程序的并发执行使得多个程序可以共享系统资源,提高系统资源利用率,还可以增加系统吞吐量。同时,程序的并发执行和资源共享也使得系统环境变得非常复杂,不像顺序执行那么简单,它产生了新的特征。

(1) 失去了封闭性。并发执行的多个程序共享系统资源,因而这些资源的状态不再仅由某一个程序所决定,而是受到并发程序的共同影响。假设程序 A、B 共享变量 n ,程序 A 能修改 n 的值,而程序 B 只是取来用。在并发环境下,由于程序 A 的运行速度快慢不同,程序 B 可能在没有被修改前读取 n ,也可能在 n 被修改后读取它,从而产生了两个不同的运行结果,造成程序结果不唯一,程序失去封闭性。

(2) 失去了可再现性。程序是指令的有序集合,是“静态”的概念,而“计算”是指令序列在处理机上的执行过程,是“动态”的概念。在并发执行中,一个共享程序可能被多个用户作业调用,从而产生多个“计算”结果,造成程序每次执行的结果可能不同,程序失去了可再现性。

(3) 并发程序间的制约性。在并发环境下,程序不再是顺序连贯执行,而是走走停停的特征。程序间不但会竞争共享资源,还可能相互协作,从而产生了相互间的制约关系。由于竞争资源导致的关系叫间接制约关系,也叫互斥关系;由于相互协作所导致的关系叫直接制约关系,也叫同步关系。

程序在并发执行时,其能否正常工作不仅与自身的正确性有关,而且与它在执行过程中能否与相关程序实施正确的同步和互斥关系有关。所以当并发执行时,解决程序间的同步和互斥问题是十分重要的。

10.1.3 程序的并行执行

和并发执行不同,程序的并行执行是指同一时刻,多个程序可以同时执行。在多处理机系统中,可以让多个进程,或同一进程内的多条线程做到真正意义上的同时执行,它们之间不需要排队(这里说的是理想情况下,因为系统中进程(线程)的数量可能超出处理机的数量,这时依然需要排队)。在这种情况下,多个程序才能达到真正意义上的“同时”执行,即并行执行。

除了多处理机系统,程序的并行执行还可能是多台计算机上的部署执行。

10.2 Goroutine

程序的并发执行基本是由操作系统提供的,很少有在语言层面就支持并发特性的,而Go语言就可以在语言层面支持并发模式的程序。

10.2.1 操作系统提供的并发基础

从操作系统层面来讲,程序并发执行的基础是进程(Process),线程(Thread)和协程(Coroutine)。

1. 进程

进程是在并发环境下,程序的一次动态执行过程。它由进程控制块(PCB)、程序和数据三部分组成,进程在它的生命周期内可能处于执行、就绪、阻塞三种基本状态。

在多任务操作系统中,多个进程可以并发执行,而且进程是系统资源分配的基本单位。系统中每个进程都有自己的内存映像区,且互不影响,所以管理简单,但缺点是系统开销大。所以,系统能同时创建的进程数量是有限的,不能太多。

2. 线程

由于进程的系统开销大,操作系统的设计者又提出了更小的能独立运行的单位——线程,试图用它来提高系统内程序并发执行的程度,从而进一步提高系统的吞吐量。

在操作系统中,线程是由进程创建的,所以它继承了进程的部分资源,且具有进程的一些基本特征。所以多个线程之间也可以并发执行,且比进程的系统开销小。但是,和进程一样,线程依然是由系统内核管理的,所以在高并发模式下,系统能创建的线程数量依然有限,效率也并不高。

3. 协程

协程本质上是一种用户态线程,不需要操作系统进行抢占式调度,而且在真正的实现中寄存于线程中。因此,协程系统开销极小,可以有效提高线程任务的并发性,避免高并发模式下线程的缺点。协程的最大优势在于其“轻量级”,可以轻松创建上百万个而不会导致系统资源衰竭,而系统最多能创建的进程、线程的数量却少得多。

使用协程的优点是编程简单,结构清晰。但缺点是需要语言的支持,如果语言不支持,则需要用户在程序中自行实现调度。目前,原生支持协程的语言还很少。

10.2.2 Goroutine 的定义

多数语言在语法层面并不能直接支持协程,而是通过库的方式支持,但这种方式功能也并不完整,比如仅提供协程的创建、销毁与切换能力。如果在这样的协程中调用一个同步I/O操作,比如网络通信、本地文件读写,那么都会阻塞其他并发执行的协程,从而无法真正达到协程的高并发性。

Go 语言在语言级别支持轻量级线程,叫做 Goroutine。Go 语言标准库提供的所有系统调用操作(包括同步 I/O 操作),都会让出处理机给其他 Goroutine。这使得轻量级线程的切换管理不依赖于系统的进程和线程,也不依赖于 CPU 的核心数量。

Goroutine 是 Go 语言运行库的功能,不是操作系统提供的功能,Goroutine 不是用线程实现的。Goroutine 就是一段代码,一个函数入口,以及在堆上为其分配的一个堆栈。因为节省了频繁创建和销毁线程的开销,所以它相对于进程、线程系统开销非常小,是轻量级的。可以很轻松地创建上百万个 Goroutine,但它们并不是被操作系统所调度执行。

10.2.3 Goroutine 的创建

Goroutine 是 Go 语言中的轻量级线程实现,由 Go 运行时管理(Runtime)。在一个函数调用前加上关键字“go”,这次调用就会在一个新的 Goroutine 中并发执行。当被调用的函数返回时,这个 Goroutine 也就自动结束了。需要注意的是,如果这个函数有返回值,那么这个返回值会被丢弃。

在 Go 语言中,可以使用关键字“go”创建并发执行的 Goroutine。基本格式如下:

```
go func()
```

例如:

```
func test() {  
    fmt.Println("Go...")  
}  
  
func main() {  
    for i := 0; i < 10; i++ {  
        go test()  
    }  
}
```

在上面的例子中,在一个 for 循环中一共调用了 10 次 test() 函数,它们是并发执行的。可是在编译执行上述代码时,会发现显示器没有任何输出信息,要解释这种现象,首先要了解 Go 语言程序的执行机制。

Go 程序从初始化 main package 并执行 main() 函数开始,当 main() 函数返回时程序退出,并且程序并不等待其他 Goroutine 结束。对于上述例子,main() 函数启动了 10 个 Goroutine,然后就直接返回退出了。而被启动的执行 test() 函数的 10 个 Goroutine 并没有来得及执行,所以程序没有任何输出结果。

在多进程或多进程编程时,操作系统解决上述问题的方法是,让父进程等待线程执行结束后再退出,比如使用 WaitForSingleObject 之类的调用,来等待所有线程执行完毕。而 Go 语言有自己特有的解决方式,那就是 Channel(通道)。Channel 可以在 Goroutine 之间进行通信,这样 main() 函数就可以知道 Goroutine 何时退出。通过 Channel,main() 函数就可以等待所有 Goroutine 都退出了自己再退出。

所以,在使用 Go 语言设计并发程序时,通常是 Goroutine 和 Channel 配合使用,二者不

可缺其一。

10.3 Channel

在并发环境中,程序间并不只是简单的并发执行,而是由于并发执行而导致的程序间的直接制约关系或间接制约关系。而要解决好程序间的制约关系,操作系统提供了许多机制,比如锁原语操作、信号量机制、共享内存、消息通信机制等。不管采用哪一种机制,都要在并发执行的程序间交换数据、进行通信。

10.3.1 程序间的并发通信

前面讲到,并发程序间的最大问题是通信,而最常见的并发通信模型是共享内存(Shared Memory)和消息机制(Message Communication Mechanism)。

1. 共享内存

共享内存是指多个并发单位分别保存对同一个数据的引用,实现对该数据的共享。被共享的数据可以有多种形式,比如内存数据块、磁盘文件、网络数据等,在实际应用中最常见的就是内存数据块。

多个并发单位在同时访问共享内存时,必须使用互斥锁等相关机制,以保证对共享内存的互斥访问。这就造成了在多个并发单位间使用共享内存通信时,程序结构往往比较复杂,程序逻辑结构难于控制等问题。

2. 消息机制

Go语言是以并发编程作为语言的最核心优势,所以在处理程序并发模型时,它不再采用共享内存作为并发单位之间的通信手段,而是以消息机制作为主要通信方法。

消息机制规定每个并发单位是自包含的、独立的个体,并且都有自己的变量,这些变量不能在不同的并发单位之间共享。每个并发单位的输入、输出只有一种,那就是消息。这有点类似于进程的概念,每个进程都不会被其他进程打扰,它只做好自己的工作就行了。不同进程间靠消息进行通信,而不会共享内存数据。

10.3.2 Channel 简介

Go语言提供的消息通信机制被称为Channel,它类似于单双向数据管道(Pipe),用户可以使用Channel在两个或多个Goroutine之间传递消息。Channel从设计上确保同一时刻只有一个Goroutine能从中接收数据,这就避免了使用互斥锁的问题。另外,Channel中数据的发送和接收都是原语操作,不会中断,只会失败。

Channel是进程内的通信方式,因此通过Channel传递对象的过程和调用函数时的参数传递行为比较一致,当然也可以传递指针等。如果需要跨进程进行通信,一般建议使用分布式系统的方法来解决,比如使用网络套接字(Socket)或者HTTP等通信协议。Go语言对网络通信也有非常完善的支持,这将在第11章中讲解。

10.3.3 Channel 声明和初始化

在 Go 语言中,Channel 是引用类型,也是类型相关的,也就是说一个 Channel 只能传递一种类型的值,这个类型需要在声明 Channel 时指定。Channel 的一般声明形式如下:

```
var chanName chan ElementType
```

从上式可以看出,Channel 的声明格式和一般变量声明基本相同,只是在类型前加了个关键字“chan”。ElementType 指定 Channel 所能传递的元素类型。

例如:

```
var ch chan int
```

该例中,ch 是一个可以传递 int 类型的 Channel。

Channel 除了可以传递基本类型的数据,还可以作为 Array、Slice 或 Map 的元素。

例如:

```
var chs [10]chan int
```

该例中,chs 是一个包含 10 个可传递 int 类型数据的 Channel。

还可以使用 make() 函数直接声明并初始化 Channel,例如:

```
ch := make(chan int)
```

该例中声明并创建了 ch,并为其分配了内存。

10.3.4 数据接收和发送

Channel 的主要用途是在不同的 Goroutine 之间传递数据,它使用通道运算符“<-”接收或者发送数据,将一个数据发送(写入)至 Channel 的语法如下:

```
ch <- value
```

向 Channel 写入数据通常会导致程序阻塞(Block),直到有其他 Goroutine 从这个 Channel 中读取数据。从 Channel 中接收(读取)数据的语法如下:

```
value := <- ch
```

如果 Channel 之前没有写入数据,那么从 Channel 中读取数据也会导致阻塞,直到 Channel 中被写入数据为止。

例 10-1 Goroutine 和 Channel 使用举例。

```
1 //Goroutine 和 Channel 使用举例
2 package main
```

```

3
4 import(
5     "fmt"
6     "math/rand"
7 )
8 func Test(ch chan int) {
9     ch <- rand.Int() //向 Channel 中写入一个随机数
10    fmt.Println("Go...")
11 }
12 func main() {
13     chs := make([]chan int, 10)
14     for i := 0; i < 10; i++ {
15         chs[i] = make(chan int)
16         go Test(chs[i]) //启动 10 个 Goroutine
17     }
18     for _, ch := range chs {
19         value := <- ch //阻塞等待退出信号
20         fmt.Println(value)
21     }
22 }

```

编译并运行该程序,输出结果为:

```

Go...
134020434
1597969999
1721070109
2068675587
1237770961
220031192
2031484958
583324308
958990240
413002649

```

在例 10-1 中,定义了一个包含 10 个 Channel 的数组 chs,同时在 main() 函数中启动了 10 个执行函数 Test() 的 Goroutine,并把数组中的每个 Channel 分配给不同的 Goroutine。在每个 Goroutine 的 Test() 函数执行后,通过 `ch <- rand.Int()` 语句向对应的 Channel 中写入一个随机数。在这个 Channel 被读取前,这个操作是阻塞的。

在所有的 Goroutine 启动完成后,通过语句 `value := <- ch` 从 10 个 Channel 中依次读出数据。在对应的 Channel 写入数据前,这个操作也是阻塞的。这样,就用 Channel 实现了类似锁的功能,从而保证了所有 Goroutine 完成后 main() 函数才返回。

在使用 Go 语言开发应用程序时,经常会遇到需要实现条件等待的问题,这时就可以使用 Channel 进行处理。对 Channel 的熟练使用,才能真正理解和掌握 Go 语言并发编程。

10.3.5 Channel 的关闭和迭代器

关闭 Channel 非常简单,直接使用 Go 语言提供的内置函数 `close()` 即可。关闭 Channel 的操作语句如下:

```
close(chanName)
```

在关闭了一个 Channel 之后,往往用户还需判断 Channel 是否被关闭,这时可以在读取 Channel 的时候使用多重返回值的方式,例如:

```
value, ok := <- ch
```

这个用法与 Map 中按键值获取 value 的过程比较类似,只需要看第二个 bool 返回值即可。如果返回值是 false 则表示 Channel 已被关闭,否则主函数还要继续阻塞接收或者发送。

对 Channel 的读取操作还可以使用 range 迭代器来完成,range 操作直至 Channel 关闭(Close)方才终止循环。另外,在 Go 语言中,还经常把创建 Goroutine 和 Channel 的工作放在一个匿名函数中来完成。

例 10-2 Channel 迭代操作。

```
1 //Channel 迭代操作
2 package main
3
4 import(
5     "fmt"
6 )
7
8 func main() {
9     ch := make(chan int)
10    go func() { //匿名函数
11        for i := 0; i < 5; i++{
12            ch <- i
13        }
14        close(ch)
15    }()
16    for value := range ch { //range 迭代器
17        fmt.Println(value)
18    }
19 }
```

编译并运行该程序,输出结果为:

```
0
1
2
3
4
```

在例 10-2 的第 14 行,如果不使用 `close()` 函数关闭 `ch`,则会引发接收端“throw: all goroutines are asleep-deadlock!”错误。

接收方还可以使用另外一种方式代替 `range` 接收数据,并判断 Channel 关闭状态,代码如下:

```
func main() {
    ch := make(chan int)
    go func() {
        for i := 0; i < 5; i++ {
            ch <- i
        }
        close(ch)
    }()
    for {
        if value, ok := <- ch; ok {
            fmt.Println(value)
        } else {
            break
        }
    }
}
```

只有发送端(另一端正在等待接收)才能关闭 Channel,只有接收端才能获得关闭状态。`Close` 调用不是必需的,但如果接收端使用 `range` 或者循环接收数据,就必须调用 `Close`,否则就会导致“throw: all goroutines are asleep-deadlock!”错误。

10.3.6 单向 Channel

前面例子中列举的通道既能发送数据,也能接收数据,被称为双向通道(Duplex-channel)。还可以将 Channel 指定为单向通道(Simplex-channel),即只能接收,或只能发送。在将一个 Channel 变量传递到一个函数时,可以通过将其指定为单向 Channel 变量,从而限制该函数对此 Channel 的操作,比如只能从此 Channel 读,或只能往该 Channel 写。

只能接收的 Channel 变量定义形式如下:

```
var chanName chan <- ElementType
```

只能发送的 Channel 变量定义形式如下:

```
var chanName <- chan ElementType
```

在定义了单向 Channel 后,还要对其初始化。在 Go 语言中,Channel 是引用数据类型,也是一个原生数据类型,因此不仅支持被传递,还支持类型转换。所以,单向 Channel 可以由一个已定义的双向(正常)Channel 转换而来。

例如:

```
ch := make(chan int)
chRead := <- chan int(ch)
chWrite := chan <- int(ch)
```

该例中基于 ch, 通过类型转换初始化了两个单向 Channel: chRead 是一个单向读 Channel, chWrite 是一个单向写 Channel。

例 10-3 单向 Channel。

```
1 //单向 Channel
2 package main
3
4 import(
5     "fmt"
6 )
7 func Recv(ch<- chan int, lock chan<- bool) {
8     for value := range ch {
9         fmt.Println(value)
10    }
11    lock<- true
12 }
13 func Send(ch chan<- int) {
14     for i := 0; i < 5; i++{
15         ch<- i
16     }
17     close(ch)
18 }
19 func main() {
20     ch := make(chan int)           //双向 Channel 可转换为单向 Channel
21     lock := make(chan bool)
22     go Recv(ch, lock)              //只能从 ch 接收的 Goroutine
23     go Send(ch)                   //只能向 ch 发送的 Goroutine
24     <- lock
25 }
```

编译并运行该程序, 输出结果为:

```
0
1
2
3
4
```

在例 10-3 中, ch 是一个正常的双向 Channel, 它可以转换为单向 Channel。函数 Send() 的参数为只写 Channel, 它可以向 ch 写入数据, 函数 Recv() 的第一个参数为只读 Channel, 它可以接收 Send() 写入的数据, 然后打印出来。在 main() 函数中创建了两个 Goroutine, 一个用于执行 Send() 函数, 另一个用于执行 Recv() 函数。这时, Send() 和 Recv() 是并发执行的, 所以源代码中第 22、23 条语句的执行次序不会影响程序运行结果。

10.3.7 异步 Channel

前面的举例创建的都是不带 Buffer 的 Channel, 这种做法只适用于传递单个数据的应用场合, 而对需要持续传输大量数据的应用场合就不适用了。对于在 Goroutine 间传输大量数据的应用, 可以使用异步通道 (Asynchronous-channel), 从而达到消息队列的效果。

异步 Channel, 就是给 Channel 设定一个 Buffer 值。在 Buffer 未写满的情况下, 不阻塞发送操作; 在 Buffer 未读完之前, 不阻塞接收操作。这里的 Buffer 是指被缓冲的数据对象的数量, 而不是内存大小。

要创建一个带 Buffer 的 Channel, 只需要在调用 `make()` 函数时, 将缓冲区的大小作为第二个参数传入即可。

例如:

```
ch := make(chan int, 1024)
```

该例创建了一个大小为 1024 的 `int` 类型 Channel, 即使没有读取方, 写入方也可以一直往 Channel 中写入数据, 在缓冲区被写满之前都不会阻塞。

从带 Buffer 的 Channel 中读取数据, 可以使用与常规非缓冲 Channel 完全一致的方法, 但一般是使用 `range` 来实现更为简洁的循环读取。

例 10-4 异步 Channel。

```
1 //异步 Channel
2 package main
3
4 import(
5     "fmt"
6     "time"
7 )
8 func Worker(sem chan int, lock chan bool, id int) {
9     sem <- 1 //相当于对同步信号量进行 P 原语操作
10    fmt.Println(time.Now().Format("04:05"), id)
11    time.Sleep(1 * time.Second)
12    <- sem //相当于对同步信号量进行 V 原语操作
13    if id == 9 {
14        lock <- true
15    }
16 }
17 func main() {
18     ch := make(chan int, 2)
19     lock := make(chan bool)
20     for i := 0; i < 10; i++ {
21         go Worker(ch, lock, i)
22     }
23     <- lock
24 }
```

编译并运行该程序, 输出结果为:

```
58:18 0
58:18 1
58:19 2
58:19 3
58:20 4
```

```
58:20 5
58:21 6
58:21 7
58:22 8
58:22 9
```

在例 10-4 中,实现了一个使用异步 Channel 模仿同步信号量(Semaphore)的功能,用来限制主函数中并发执行的 Goroutine 的数量。该例中变量 ch 的 Buffer 值被设为 2,所以主函数能同时执行 Goroutine 的数量为 2。通过程序的运行结果,证明了这一点。比如 58 : 18 这个时间段内执行了 Goroutine 0 和 1,58 : 19 这个时间段内执行了 Goroutine 2 和 3。

10.4 Select 机制和超时机制

在 Go 语言中,Select 机制主要用于解决通道通信中的多路复用问题。因为通道的接收操作往往是阻塞式的,所以 Select 机制还经常和超时机制配合使用,将阻塞式的通信转换为非阻塞式,以提高系统通信效率。

10.4.1 Select 机制

在 UNIX 系统中,Select 机制已被引入。系统可以通过调用 select() 函数来监控一系列文件句柄,一旦其中一个文件句柄发生 I/O 操作,该 select() 函数调用就会被返回。在网络编程中,Select 机制也被用来监听多个非阻塞套接字(Socket),以实现高并发的服务器程序。Go 语言则直接在语言级别,使用“select”关键字处理异步 I/O 问题。

如果有多个 Channel 需要监听,就可以使用 Select 机制,随机处理一个可用的 Channel。Select 的用法和 Switch 语句非常类似,由“select”开始一个新的选择块,每个选择条件由“case”语句来描述。与 Switch 语句可以使用任何形式条件表达式相比,Select 机制有比较多的限制,其中最大的一条是每个“case”语句必须是一个 I/O 操作。Select 机制的基本结构如下:

```
select {
    case <- chan1:
        //如果 chan1 成功读取数据,则进行该 case 处理语句.
    case <- chan2:
        //如果 chan2 成功读取数据,则进行该 case 处理语句.
        :
    default:
        //如果上面都没有成功,则进入 default 处理流程.
}
```

可以看出,Select 不像 Switch 语句,后面并不带判断条件,而是直接检测 case 语句。每个 case 语句都必须是一个面向 Channel 的操作。

例 10-5 Select 机制。

```
1 //Select 机制
2 package main
3
4 import(
5     "fmt"
6 )
7 func main() {
8     ch1 := make(chan int)
9     ch2 := make(chan string)
10    lock := make(chan bool)
11    go func() {
12        for {
13            select {
14                case value, ok := <- ch1:
15                    if !ok {
16                        lock <- true
17                        break
18                    }
19                    fmt.Println("ch1 = ", value)
20                case value, ok := <- ch2:
21                    if !ok {
22                        lock <- true
23                        break
24                    }
25                    fmt.Println("ch2 = ", value)
26            }
27        }
28    }()
29    ch1 <- 100
30    ch2 <- "Golang"
31    ch2 <- "Go"
32    ch1 <- 200
33    close(ch1)
34    close(ch2)
35    <- lock
36 }
```

编译并运行该程序,输出结果为:

```
ch1 = 100
ch2 = Golang
ch2 = Go
ch1 = 200
```

在例 10-5 中,Select 在接收端进行监听,当然 Select 也可以用于发送端,如果有其他 Goroutine 在运行,还可以使用空“select{}”阻塞 main()函数,避免进程终止。

通过以上知识和事例可以看出,Select 机制的基本过程如下:

- (1) 当所有被监听 Channel 中都无数据时, Select 会一直等到其中一个有数据为止。
- (2) 当多个被监听 Channel 中都有数据时, Select 会随机选择一个 case 执行。
- (3) 当所有被监听 Channel 中都无数据,且 default 子句存在时,则 default 子句会被执行。
- (4) 如果想持续监听多个 Channel,需要使用 for 语句协助。

10.4.2 超时机制

在前面所有举例中,所有对 Channel 操作的错误问题都被忽略了,没有进行错误处理的程序显然是不安全的。即在向 Channel 写数据时发现已满,或从 Channel 读数据时发现为空,如果不正确处理这些问题,很可能会导致整个 Goroutine 死锁。在 Go 语言并发编程的通信过程中,所有错误处理都由超时机制来完成。

超时机制是一种解决通信死锁的机制,通常会设置一个超时参数,通信双方如果在设定的时间内仍然没处理完任务,则该处理过程会立即被终止,并返回对应的超时信息。

Go 语言没有提供直接的超时处理机制,但可以利用 Select 机制解决超时问题。因为 Select 的特点是只要其中一个 case 已经完成,程序就会继续往下执行,而不会考虑其他 case 的执行情况。基于此特性,就可以使用 Select 为 Channel 实现超时处理功能。

例 10-6 使用 Select 实现 Channel 超时机制。

```
1 //使用 Select 实现 Channel 超时机制
2 package main
3
4 import(
5     "fmt"
6     "time"
7 )
8 func main() {
9     ch := make(chan int)
10    timeout := make(chan bool,1)
11    go func() {
12        time.Sleep(5 * time.Second)
13        timeout <- true
14    }
15    select {
16    case <- ch:
17    case <- timeout:
18        fmt.Println("timeout...")
19        break
20    }
21 }
```

编译并运行该程序,输出结果为:

```
timeout...
```

在例 10-6 中,首先实现并执行了一个匿名超时等待函数,该等待 5s 后向 Channel 变量 timeout 中写入 true。然后在 Select 的一个 case 中接收 timeout。如果其他 case 语句中的

Channel 都没有在 5s 之内接收到数据,则只有 timeout 接收到数据,则意味着主程序超时退出。

另外,在 Go 语言的 time 包中还提供了 After、Tick 等函数,它们也可以返回计时器 Channel,利用它们也可以实现超时机制。

使用 After 函数,例 10-6 的实现代码如下:

```
func main() {
    ch := make(chan int)
    lock := make(chan bool)
    go func() {
        for {
            select {
            case <- ch:
            case <- time.After(5 * time.Second):
                fmt.Println("timeout...")
                lock <- true
                break
            }
        }
    }()
    <- lock
}
```

10.5 Runtime Goroutine

在执行一些高并发的计算任务时,为了尽量利用高性能服务器多核心的特性,将计算任务并行化,从而达到降低总计算时间的目的,在 Go 语言的 Runtime 包中提供了几个与 Goroutine 相关的函数,例如 Gosched()、NumCPU()、Goexit()等,这些函数可以对 Goroutine 进行控制,或者获取 CPU 信息,以期达到并行任务处理的目的。

10.5.1 出让时间片

在设计并发任务时,用户可以在每个 Goroutine 中控制何时主动让出时间片给其他 Goroutine,这可以使用 Gosched()函数来实现。Gosched()类似于 C# 或 Python 中的 Yield(函数迭代器),让出当前 Goroutine 的执行权限,调度器会安排其他等待的任务去运行,并在下轮某个时间片再从该位置恢复执行。

例 10-7 出让时间片。

```
1 //出让时间片
2 package main
3
4 import(
5     "fmt"
6     "time"
```

```
7     "runtime"
8 )
9 func main() {
10     go func() {
11         for i := 0; i < 5; i++{
12             if i == 2 {
13                 runtime.Gosched()
14             }
15             fmt.Println("Goroutine1: ", i)
16         }
17     }()
18     go func() {
19         fmt.Println("Goroutine2")
20     }()
21     time.Sleep(5 * time.Second)
22 }
```

编译并运行该程序,输出结果为:

```
Goroutine1: 0
Goroutine1: 1
Goroutine2
Goroutine1: 2
Goroutine1: 3
Goroutine1: 4
```

在例 10-7 中,Goroutine1 一共要执行 5 个工作步骤,当执行到第 2 个工作步骤时调用了 Gosched()函数,则 Goroutine1 将让出时间片给 Goroutine2。当 Goroutine2 获取执行权限执行完毕后,Goroutine1 则返回到第 3 个工作步骤继续执行。

10.5.2 获取 CPU 核心数和任务数

有时为了将多个并发执行的 Goroutine 分配给不同的 CPU 核心去完成,用户就需要知道 CPU 核心的具体数目。为此,runtime 包提供了 NumCPU()函数可以完成这个任务。

为了观察系统任务调度情况,还可以使用 NumGoroutine()函数返回正在执行和排队的任务总数。

例 10-8 统计核心数和任务数。

```
1 //统计核心数和任务数
2 package main
3
4 import(
5     "fmt"
6     "time"
7     "runtime"
8 )
9 func main() {
```



```

10     fmt.Println("CPU number: ",runtime.NumCPU())
11     fmt.Println("Goroutines start: ",runtime.NumGoroutine())
12     for i := 0; i < 5; i++{
13         go func(n int) {
14             fmt.Println(n,runtime.NumGoroutine())
15         }(i)
16     }
17     time.Sleep(5 * time.Second)
18     fmt.Println("Goroutines over: ",runtime.NumGoroutine())
19 }

```

编译并运行该程序,输出结果为:

```

CPU number: 2
Goroutines start: 2
0 8
1 7
2 6
3 5
4 4
Goroutines over: 3

```

通过运行结果可以发现,系统的 CPU 核心数为 2,在 main()函数创建 Goroutine 之前, Goroutine 总数为 2(main 和 GC Heap 管理总是在运行)。

10.5.3 终止当前 Goroutine

如果要强行终止一个 Goroutine 的执行,可以调用 Goexit()函数来完成。Goexit()将终止整个堆栈链,并在内层退出,但是 defer 语句仍然或被执行。

例 10-9 强行终止 Goroutine。

```

1 //强行终止 Goroutine
2 package main
3
4 import(
5     "fmt"
6     "time"
7     "runtime"
8 )
9 func main() {
10     go func() {
11         defer fmt.Println("Goroutine1 defer...")
12         for i := 0; i < 10; i++{
13             if i == 5 {
14                 runtime.Goexit()
15             }
16             fmt.Println("Goroutine1: ",i)
17         }

```

```
18     }()  
19     go func() {  
20         fmt.Println("Goroutine2")  
21     }()  
22     time.Sleep(5 * time.Second)  
23 }
```

编译并运行该程序,输出结果为:

```
Goroutine1: 0  
Goroutine1: 1  
Goroutine1: 2  
Goroutine1: 3  
Goroutine1: 4  
Goroutine1 defer...  
Goroutine2
```

在例 10-9 中,Goroutine1 一共要执行 10 个工作步骤,当执行到第 5 个工作步骤时调用了 Goexit() 函数,则 Goroutine1 被强行终止执行。Goroutine2 则获取权限得到执行。

小结

本章主要介绍了 Go 并发程序设计思想,以及程序并发执行机制,还介绍了程序、作业、进程、线程和协程这几个程序执行单位之间的区别与联系。另外,重点介绍了 Go 语言支持的并发执行单位协程“goroutine”,介绍了通道的概念和类型,包括如何利用通道在不同的 goroutine 之间进行通信。最后介绍了 Select 机制和超时机制,还介绍了 Runtime 任务调度机制。

学习这一章内容,对缺少计算机组成原理和操作系统知识的读者来说可能有一定的难度,所以建议读者要准备一些上述知识。另外,通过这一章的学习,了解了这门编程语言为什么叫“Go”,因为“Go”就是创建“goroutine”的关键字。所以,Go 语言最大的特点就是 from 语言层面就支持并发设计,这是它具有活力和生命力的最大特征。因为现代程序设计技术的发展方向就是“平台都是多核心的、任务都是高并发的”,比如正在迅猛发展的云计算。所以 Go 语言其实就是顺应现代计算机发展的一个必然产物,它的目标是尽可能在静态语言和动态语言之间取得一个最大的优势互补。

习题

- OSI 参考模型共有 7 层,从下到上依次为物理层、数据链路层、网络层、传输层、会话层、应用层。OSI 参考模型各层的主要功能是:
- (1) 物理层是 OSI 参考模型中的最底层,主要产生原始比特信号,并为数据链路层提供传输。
 - 10.1 在多任务操作系统中,进程和线程的执行具有并发性,并发是指 _____。
 - 10.2 Go 语言在语言级别支持轻量级线程,叫做 _____。
 - 10.3 在 Go 语言中,一个函数调用前加上关键字“go”,这次调用就会在一个新的 Goroutine 中并发执行。下面的程序需要创建 10 个 test() 的 Goroutine,将程序补充完整。

```
func test() {
    fmt.Println("Go...")
}

func main() {
    for i := 0; _____; i++{
        _____ test()
    }
}
```

10.4 Go 语言提供的消息通信机制被称为_____,它类似于单双向数据管道(Pipe), 用户可以使用它在两个或多个 Goroutine 之间传递消息。

10.5 通常定义的通道既能发送数据,也能接收数据,被称为_____通道。还可以将 Channel 指定为单向通道,即只能接收,或只能发送。定义一个只能接收的整型通道的语句为:_____,定义一个只能发送的整型通道的语句为:_____。

10.6 阅读下面的程序:

```
func main() {
    ch := make(chan int)
    go shower(ch)
    for i := 0; i < 10; i++{
        ch <- i
    }
}

func shower(c chan int) {
    for {
        j := <- c
        fmt.Printf("%d ", j)
    }
}
```

上述程序的运行结果为:_____。

10.7 在第 6 章和第 7 章中分别使用数组和递归函数实现了 Fibonacci 数列的输出,这里要求试使用 Channel 实现 Fibonacci 数列输出。

第11章

Go 网络编程

11.1 Go 网络编程简介

Go 语言的标准库里提供了 net 包,它支持基于网络层(IP 层)、传输层(TCP/UDP 层)以及应用层(如 HTTP、FTP、SMTP)的网络通信。要学好网络编程技术,除了要掌握这些库函数外,还要了解一些和网络编程相关的计算机网络基础知识。

11.1.1 计算机网络概念和体系结构

计算机网络就是使用通信链路,将分布在不同地理区域内具有独立自主工作能力的计算机、终端及其附属设备互连起来,由网络操作系统管理,进行相互通信和资源共享的系统。

计算机网络是一个十分复杂的系统,将其分解为若干个容易处理的层次(Layer),然后“分而治之”,这种结构化设计方法是工程设计中常见的手段。在计算机网络技术发展的历史过程中,ISO/OSI 参考模型和 TCP/IP 体系结构是网络分层设计的两个典型代表。

1. ISO/OSI 参考模型

为了实现不同网络体系之间的相互兼容,国际标准化组织 ISO 于 1981 年制定了开放系统互连参考模型(Open System Interconnection/Reference Mode, OSI/RM)作为网络体系结构的国际标准。

OSI 参考模型共有 7 层,从下到上依次为物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。OSI 参考模型各层的主要功能是:

- (1) 物理层是 OSI 参考模型中的最底层,主要产生原始比特信号,并为数据链路层提供传输比特流的一个物理连接。
- (2) 数据链路层将比特流封装成一个完整的通信单位“帧”,在相邻节点之间进行可靠通信,并提供差错校验、流量控制等功能。
- (3) 网络层的主要功能是实现网络互连,并在不同网络之间提供路由功能。

(4) 传输层负责端到端的通信,既是 7 层模型中负责数据通信的最高层,又是面向网络通信的低三层和面向信息处理的高三层之间的中间层,它主要提供面向连接的通信服务和无连接的通信服务。

(5) 会话层在传输层提供的服务上,加强了会话管理、同步和活动管理等功能。会话层的主要任务是为用户提供特定应用进程的连接服务,并控制、管理和同步会话层实体之间的对话。

(6) 表示层的主要功能为语法转换、语法协商和连接管理,由于各种计算机都可能各有各的数据描述方法,所以不同类型计算机之间交换的数据,一般需经过格式转换才能保证其意义不变。

(7) 应用层是 OSI 参考模型的最高层,直接面向用户。它为用户访问 OSI 提供必要的手段和服务,比如 DNS、FTP、Telnet、DHCP、SMTP 服务等。

2. TCP/IP 体系结构

虽然 OSI 参考模型是计算机网络发展的一种国际标准,但由于 Internet 在全球的飞速发展,因此 Internet 所使用的 TCP/IP 体系结构就在计算机网络领域占有特别重要的地位,成为计算机网络事实上的工业标准。TCP/IP 参考模型共分为 4 个层次,分别是网络接口层、网际层、传输层和应用层。

在 TCP/IP 的不断发展过程中也吸收了 OSI 标准中的概念及特征,OSI/RM 参考模型的制定,也参考了 TCP/IP 协议簇及其分层体系结构的思想。OSI/RM 参考模型和 TCP/IP 体系结构的对比如图 11-1 所示,二者的区别如下:

(1) TCP/IP 参考模型不存在 OSI 中的表示层和会话层,表示层和会话层的功能实际上可以由应用层和传输层完成。

(2) TCP/IP 参考模型中不存在数据链路层和物理层,取而代之的是网络接口层(有的文献称为主机至网络层)。

(3) TCP/IP 模型是对协议簇的抽象,换句话说,先有协议后有模型。这是 TCP/IP 模型与 OSI 另一个主要不同之处。

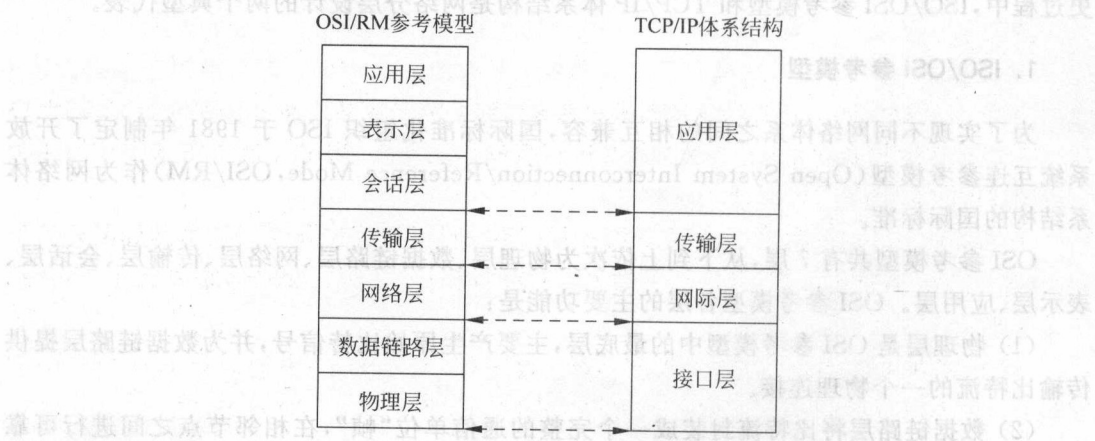


图 11-1 OSI/RM 与 TCP/IP 层次对比图

3. TCP/IP 协议簇

TCP/IP 协议簇是 Internet 的标准连接协议,主要包括 IP、TCP 和 UDP。

(1) IP。IP(Internet Protocol)是 Internet 网络层核心协议,它提供不可靠、无连接的分组传递网络服务。IP 定义了 Internet 上相互通信的计算机的 IP 地址,并通过路由选择,将数据报由一台计算机传递到另一台计算机,从而实现不同网络之间的互联。目前主要有 IPv4、IPv6 两个版本,IPv4 地址长度为 32 位,IPv6 地址长度为 128 位。

(2) TCP。TCP(Transmission Control Protocol,传输控制协议)是传输层一种面向连接的通信协议,提供可靠的数据传送。对于大量数据的传输,通常都要求有可靠的传送。它使用三次握手和滑动窗口机制来保证传输的可靠性和进行流量控制。

(3) UDP。UDP(User Datagram Protocol,用户数据报协议)是一种面向无连接的协议,它不能提供可靠的数据传输,而且 UDP 不进行差错检验,必须由应用层的应用程序实现可靠性机制和差错控制,以保证端到端数据传输的正确性。

11.1.2 网络编程基本概念

按照操作系统进程通信的观点,网络通信也属于一种进程通信,而且是不同主机上进程间的高级通信。而在一个自治系统内,进程间的主要通信方式有信号(Signal)、消息(Message)、管道(pipe)、共享内存(Shared Memory)等。

网络编程是要实现计算机网络中不同主机进程间进程通信,为此,首先要解决如何识别网络中不同主机进程的问题。在自治系统内,是用进程号(Process ID,PID)唯一标识一个进程,进程间通信就可以使用 PID 来进行。在网络环境下,PID 已不能唯一标识一个进程。例如,主机 A 赋予某进程号 100,在 B 机中也可以存在 100 号进程,因此,进程号对于网间进程通信就没有意义了。所以,网间进程通信还要解决网间进程的标识问题。

另外,由于不同的主机可能使用不同的网络协议,其工作方式不同,地址的表示格式也不同。因此,网间进程通信还要解决多种协议识别问题。为了解决上述问题,在网络编程时还需引入 IP 地址、协议端口、连接与相关等几个概念。

1. IP 地址

IP 地址是 Internet 上主机地址的数字形式,与主机的域名地址一一对应。IP 地址由 32 位的二进制数据表示,用于屏蔽各种物理网络的地址差异,通常写成用句点分开的 4 个十进制数的形式,例如 192.168.0.1。

IP 地址是网络的逻辑地址,在互联网中具有唯一标记网络的作用。它由国际网络信息中心(NIC)统一管理和分配,保证互联网上运行的设备(如主机、交换机、路由器等)不会产生地址冲突。

(1) IP 地址的组成。一个互联网包括多个网络,而一个网络又包括多台主机。因此,为了能唯一标识网络设备,在 32 位的 IP 地址中由网络号和主机号组成,如图 11-2 所示。

网络号	主机号
-----	-----

同一个物理网络上的所有主机都具有相同的网络号,网络上的每

图 11-2 IP 地址结构

一台主机(包括网络上工作站、服务器和路由器等)都有一个主机号与其对应。

用 32 位二进制数表示 IP 地址,记忆起来非常麻烦。为了方便用户的理解和记忆,通常采用点分十进制的标记方法。它将 IP 地址的 4 个字节域的每个字节都转换成相应的十进制数,中间用“.”号隔开,表示成 w. x. y. z 的形式。

例如,二进制 IP 地址

11000000 10101000 00000001 11111110

用点分十进制表示成

192.168.1.254

(2) IP 地址的分类。在互联网中,物理网络的数量是一个难以确定的因素,而不同种类的物理网络中主机的数量也相差很大,有的物理网络有成千上万台主机,而有的仅有几台主机。为了适应各种不同的物理网络规模,IP 协议将 IP 地址分成 A、B、C、D、E 5 类,它们分别使用 IP 地址的前几位加以区分。

① A 类地址。用于主机数量非常大的大型网络。A 类地址第一个字节为网络号,最高位是类别位 0,网络号的范围是 $1 \sim 2^7 - 2$ 。最后三个字节为主机号,范围是 $1 \sim 2^{24} - 2$ 。

② B 类地址。用于主机数量多的中型网络,如城域网。B 类地址前两个字节为网络号,其中前两位是类别位 10,网络号范围是 $1 \sim 2^{14} - 2$ 。最后两个字节为主机号,范围是 $1 \sim 2^{16} - 2$ 。

③ C 类地址。用于主机数量少的小型本地网络,如局域网。C 类地址前两个字节为网络号,前三位是类别位 110,网络号范围是 $1 \sim 2^{21} - 2$ 。最后一个字节为主机号,范围是 $1 \sim 2^8 - 2$ 。

④ D 类地址。是多点播送地址,用于多目的地信息的传输和作为备用。其前 4 位为类别位 1110,第一个字节的范围为 224~239,全零(0.0.0.0)地址对应于当前主机,全 1 的 IP 地址(255.255.255.255)是当前子网的广播地址。

⑤ E 类地址。以 1111 开始,该类地址保留,仅作试验和开放之用。

(3) 特殊 IP 地址。IP 地址除了可以在 Internet 中唯一标识一台主机外,还有几种特殊的表现形式。

① 网络地址。用于标识 Internet 上一个唯一的物理网络,IP 地址中网络号位保持不变,主机号位全为“0”的地址是网络地址。例如,C 类 IP 地址 202.91.120.16 的网络地址为 202.91.120.0。

② 主机地址。用于标识物理网络中一台特定的主机,IP 地址中主机号位保持不变,网络号位全为“0”的地址是主机地址。例如,C 类 IP 地址 202.91.120.16 的主机地址为 0.0.0.16。

③ 广播地址。当一个设备发送的数据能被网络中的多个设备同时收到时,这样的通信叫广播。发送设备在以广播方式通信时必须设置广播地址,广播地址可分为直接广播(Directed Broadcasting)地址和有限广播(Limited Broadcasting)地址。

- 直接广播地址,可以将数据包广播给一个已知网络号的网络中的所有主机。直接广播地址网络号位保持不变,主机号位全为“1”,例如,C 类网络地址 202.91.120.16 的直接广播地址为 202.91.120.255。
- 有限广播地址,只能用于本网广播,网络号位和主机号位全为“1”,即 255.255.255.255。实际上,有限广播将广播限制在最小的范围内。如果采用标准 IP 编址,有限广播被限制在本网之内;如果采用子网编址(见 5.2.2 节),那么有限广播将被限制在本子网之中。

④ 子网掩码地址。子网掩码地址也是一个 32 位的地址,IP 地址中主机号位全为 0,网络号位全为 1 的地址是子网掩码。子网掩码和主机地址通过“逻辑与”运算可以屏蔽主机号部分,以区别网络号和主机号。例如,C 类 IP 地址 202.91.120.16 的子网掩码是 255.255.255.0。

⑤ 回送地址。A 类 IP 地址 127.0.0.1 是一个保留地址,用于网络软件测试及本机进程间通信,这个地址叫回送地址(Loopback Address)。任何进程,一旦使用回送地址发送数据,协议软件不进行任何网络传输,立即将之返回。因此,网络号为 127 的数据报不会出现在任何网络上。

⑥ 私有地址。私有地址不需要注册,仅用于局域网内部,该地址在局域网内部是唯一的。当网络上的公用地址不足时,可以通过网络地址解析(Network Address Translation, NAT),利用少量的公用地址把大量的配有私用地址的机器连接到公用网上。根据规定,私有地址的路由信息不会在因特网上传递,从而采用私有地址作为目的地址的数据包不会在因特网上进行路由。

在 RFC1918 私有网络地址分配(Address Allocation for Private Internets)建议中,推荐的私有地址范围如下。

- A 类: 10.0.0.0~10.255.255.255。
- B 类: 172.0.0.0~172.31.255.255。
- C 类: 192.168.0.0~192.168.255.255。

2. IPv6 地址

前面介绍的 IP 地址其实是 32 位的 IPv4 地址,本教材如果不做特殊说明 IP 地址都是指 IPv4 地址。随着网络用户越来越多,早期的 IPv4 地址已经不够用了,所以 Internet 逐渐向 IPv6 过渡,随之网络地址也由 32 位的 IPv4 地址升级至 128 位的 IPv6 地址。

IPv6 与 IPv4 地址之间最明显的差别在于长度: IPv4 地址长度为 32 位,而 IPv6 地址长度为 128 位。RFC2373 中不仅解释了这些地址的表现方式,同时还介绍了不同的地址类型及其结构。IPv4 地址可以被分为两或三个不同部分(网络标识符、主机标识符,有时还有子网标识符),IPv6 地址中拥有更大的地址空间,可以支持更多的字段。

(1) IPv6 地址结构与分类。IPv6 地址结构由两个域组成: 子网前缀和接口标识符,如图 11-3 所示。子网前缀用于区分不同的地址类型,接口标识符用于标识一个链路上的接口。

子网前缀	接口标识符
------	-------

图 11-3 IPv6 地址结构

IPv6 地址有三类: 单播、组播和泛播地址,单播和组播地址与 IPv4 的地址非常类似,但 IPv6 中不再支持 IPv4 中的广播地址,而增加了一个泛播地址。

① 单播地址是指单一接口的地址,发送到单播地址的数据包被送到由该地址标识的接口。

② 组播地址是指一组接口的地址,通常分属不同节点,发送到组播地址的数据包被送到由该地址标识的每个接口。

③ 泛播地址也是指一组接口的地址,同样分属不同节点,和组播不同的是发送到泛播地址的数据包被发送到由该地址标识的其中一个接口。由于使用泛播地址的标准尚在不断完善中,所以目前不推荐使用。

(2) IPv6 地址表示方式。IPv6 地址长度 4 倍于 IPv4 地址,表达起来的复杂程度也是

IPv4 地址的 4 倍。IPv6 地址的基本表达方式是 X:X:X:X:X:X:X:X, 其中 X 是一个 4 位十六进制整数(16 位)。每一个数字包含 4 位, 每个整数包含 4 个数字, 每个地址包括 8 个整数, 共计 128 位。

例如, 下面是一些合法的 IPv6 地址:

CDCD:910A:2222:5498:8475:1111:3900:2020

1030:0:0:0:C9B4:FF12:48AA:1A2B

2000:0:0:0:0:0:0:1

请注意这些整数是十六进制整数, 其中 A~F 表示的是 10~15。地址中的每个整数都必须表示出来, 但起始的 0 可以不必表示。这是一种比较标准的 IPv6 地址表达方式, 此外还有另外两种更加清楚和易于使用的方式。

某些 IPv6 地址中可能包含一长串的 0 (就像上面的第二和第三个例子一样)。当出现这种情况时, 标准中允许用“空隙”来表示这一长串的 0。换句话说, 地址“2000:0:0:0:0:0:0:1”可以被表示为“2000::1”。地址中的“::”表示该地址可以扩展成一个完整的 128 位地址。在这种方法中, 只有当 16 位组全部为 0 时才会被两个冒号取代, 且两个冒号在地址中只能出现一次。

(3) 混合表示方式。在 IPv4 和 IPv6 的混合环境中可能有第三种方法。IPv6 地址中的最低 32 位可以用于表示 IPv4 地址, 该地址可以按照一种混合方式表达, 即 X:X:X:X:X:X:d.d.d.d, 其中 X 表示一个 16 位整数, 而 d 表示一个 8 位十进制整数。

例如, 地址“0:0:0:0:0:0:10.0.0.1”就是一个合法的 IPv4 地址。把两种可能的表达方式组合在一起, 该地址也可以表示为: “::10.0.0.1”。

(4) CIDR 表示方式。由于 IPv6 地址被分成两个部分: 子网前缀和接口标识符, 因此人们期待一个 IP 节点地址可以按照类似 CIDR 地址的方式被表示为一个携带额外数值的地址, 其中指出了地址中有多少位是掩码。即 IPv6 节点地址中指出了前缀长度, 该长度与 IPv6 地址间以斜杠区分。例如, “1030:0:0:0:C9B4:FF12:48AA:1A2B/60”这个地址中用于选路的前缀长度为 60 位。

3. 协议端口

按照 OSI 7 层协议的描述, 传输层与网络层在功能上的最大区别是传输层提供进程通信能力。从这个意义上讲, 网络通信的最终地址就不仅是主机地址了, 还包括可以描述网间进程的某种标识符。为此, TCP/IP 提出了协议端口(Protocol Port)的概念, 简称端口, 用于唯一标识网间通信进程。

端口是一种抽象的软件结构(包括一些数据结构和 I/O 缓冲区)。网间进程通过系统调用与某端口建立连接(Binding)后, 传输层传给该端口的数据都被相应进程所接收, 相应进程发给传输层的数据都通过该端口输出。在 TCP/IP 的实现中, 端口操作类似于一般的 I/O 操作, 进程获取一个端口, 相当于获取本地唯一的 I/O 文件, 可以用一般的读写原语对端口进行“读”、“写”操作。

类似于文件描述符, 每个协议端口都拥有一个叫端口号(Port Number)的整型标识符, 用于区别不同端口。端口号使用 16 位的数来表示, 取值范围是 0~65 535。由于 TCP/IP 传输层的两个协议 TCP 和 UDP 是完全独立的两个软件模块, 因此各自的端口号也相互独

立,如 TCP 有一个 255 号端口,UDP 也可以有一个 255 号端口,二者并不冲突。TCP/IP 提供的端口号共分为下面两大类。

(1) 服务器使用的端口号。TCP/IP 给服务器分配端口号采用全局分配的方式,也分为两类。第一类叫作公知端口号(Well-known Port)或系统端口号,范围为 0~1023。这些信息可以在 IANA(The Internet Assigned Numbers Authority,互联网数字分配机构)的官方网站 www.iana.org 上查到,一些常用的网络服务器和熟知端口号如表 11-1 所示。

表 11-1 常见服务器端口号

网络服务	熟知端口号	网络服务	熟知端口号
FTP	21	TFTP	69
TELNET	23	HTTP	80
SMTP	25	SNMP	161
DNS	53		

第二类叫做登记端口号,范围为 1024~49 151。这类端口号是为没有熟知端口号的服务器程序使用的,使用这类端口号必须向 IANA 登记,以防重复。

(2) 客户机使用的端口号。TCP/IP 给客户机分配端口采用本地分配的方式,又称动态连接,即进程需要访问传输层服务时,向本地操作系统提出申请,操作系统返回一个本地唯一的端口号,进程再通过合适的系统调用将自己与该端口号绑定起来,范围为 49 152~65 535。由于这类端口号仅在客户机进程运行时才动态选择,因此又叫做临时端口号(Ephemeral Port),当通信结束后,刚才使用过的客户端口号就不复存在了。

4. 连接与相关

连接是指网络中两个进程之间的通信链路,相关是指组成一个连接的两个进程的元素组。为了唯一标识网络中的一个通信进程,可以使用如下的三元组。

(协议,本地地址,本地端口号)

这样的一个三元组只指定了通信时一个连接的一半,即通信的一方,所以叫做一个半相关(half-association)。如果要完整地表示网络中进行通信的两个进程,可以使用如下的五元组来标识。

(协议,本地地址,本地端口号,远程地址,远程端口号)

需要注意的是,一个完整的网络间进程通信只能使用同一种高层协议,也就是说,不能通信的一端用 TCP,而另一端用 UDP。这样一个五元组,叫做一个相关(Association),即两个协议相同的半相关才能组合成一个可用的相关。

11.1.3 网络编程模式

网络编程模式主要有客户-服务器模式(C-S 模式)和浏览器-服务器模式(B-S 模式)两种。

1. C-S 模式

C-S 模式即客户-服务器模式,在 TCP/IP 网络应用中,绝大多数网络应用的关系属于 C-S 模式。C-S 模式主要由客户应用程序(Client)、服务器管理程序(Server)和中间件

(middleware)三个部件组成,如图 11-4 所示。客户应用程序是系统中用户与数据进行交互的部件。服务器程序负责有效地管理系统资源,如管理一个信息数据库,其主要工作是当多个客户并发地请求服务器上的相同资源时,对这些资源进行最优化管理。中间件负责连接客户应用程序与服务器管理程序,协同完成一个作业,以满足用户查询管理数据的要求。

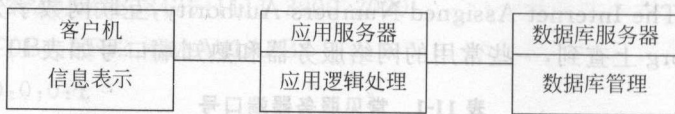


图 11-4 三层 C-S 架构

在 C-S 模式中客户机向服务器发送服务请求完全是随机的,并且可能有多个客户机同时发送请求。服务器需要随时通过熟知端口来侦听服务请求,并且要具备同时处理多个并发请求的能力,这是服务器与客户机设计中的最大区别。服务器并发处理的解决方案基本分为两种:并发服务器(Concurrent-server)与重复服务器(Interactive-server),这两种方案分别适合于不同的服务类型。

(1) 并发服务器。并发服务器采用的是工作在后台的守护进程(Daemon),当有服务请求到达时将会激活该进程来进行处理,并发服务器本身始终要处于等待并侦听的状态。当服务器接收到客户机发送的服务请求时,它根据该请求的进程号去激活子进程提供服务,而服务器自身会回到等待状态继续侦听请求。

在这种方案中,并发服务器自身被称为主(Master)服务器,它激活的子进程被称为从(slave)服务器。因此,服务器必须拥有一个全网熟知的进程地址。由于不同的从服务器可以并发、独立地处理不同客户机的服务请求,因此并发服务器比较适合于面向连接的服务类型。

(2) 重复服务器。重复服务器采用请求队列来存储到达的服务请求,并根据先到先服务的原则顺序处理服务请求。重复服务器处理客户机请求的数量受队列长度的限制,但是可以有效控制对服务请求的处理时间,因此它比较适合于无连接的服务类型。

重复服务器的工作原理是:客户机与服务器首先形成自己的半相关的三元组,然后客户机再根据服务器的熟知端口建立全相关的五元组。

在 C-S 模式中,网络服务采取的是主动请求的方式。服务器方要先启动,并根据客户机的请求提供相应的服务。具体工作步骤如下:

(1) 服务器打开一个通信通道并告知本地主机,它准备好在某一地址和端口上接收客户机的连接请求。

(2) 服务器在公有端口上进行监听,等待客户机的连接请求到达该端口。

(3) 当服务器接收到重复服务请求时,处理该请求并发送应答信号。当接收到并发服务请求时,服务器要激活一个新的进程(或线程)来处理这个客户请求。新进程(或线程)处理此客户请求,并不需要对其他请求作出应答。当通信完成后,服务器关闭服务线程,并断开与客户机的通信链路。

(4) 返回第(2)步,继续等待下一个客户的连接请求。

(5) 服务结束,关闭服务器。

在服务器准备就绪的前提下,客户机可以按如下步骤和服务器建立通信连接。

- (1) 客户机打开一个通信通道,并连接到服务器所在主机的公知端口。
- (2) 客户机向服务器发出连接请求,并等待并接收应答。
- (3) 如果服务器接收了客户机的连接请求,即通信链路建立成功,双方就可以互发数据了。
- (4) 当请求结束后,客户机关闭通信通道并终止连接。

2. B-S 模式

B-S 模式即浏览器-服务器模式,又称 B-S 架构。它是随着 Internet 技术的兴起,对 C-S 模式应用的扩展。B-S 模式以 Web 技术为主,把传统 C-S 模式中的服务器部分分解为一个数据库服务器与一个或多个应用服务器(Web 服务器),从而构成一个三层结构的客户服务器体系,如图 11-5 所示。

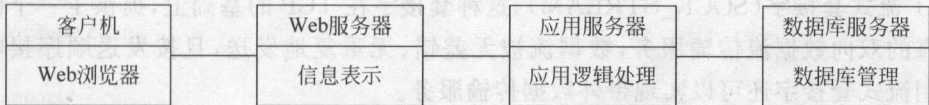


图 11-5 B-S 架构

第一层:客户机是用户与整个系统的接口。客户的应用程序精简到一个通用的浏览器软件,如 Netscape Navigator、微软公司的 IE 等。浏览器将 HTML 代码转化成图文并茂的网页。网页还具备一定的交互功能,允许用户在网页提供的申请表上输入信息提交给后台,并提出处理请求。这个后台就是第二层的 Web 服务器。

第二层:Web 服务器将启动相应的进程来响应这一请求,并动态生成一串 HTML 代码,其中嵌入处理的结果,返回给客户机的浏览器。如果客户机提交的请求包括数据的存取,Web 服务器还需与数据库服务器协同完成这一处理工作。

第三层:数据库服务器的任务类似于 C-S 模式,负责协调不同的 Web 服务器发出的 SQL 请求,管理数据库。

在这种结构下,用户工作界面是通过 IE 浏览器来实现的。B-S 模式最大的好处是运行维护比较简便,能实现不同的人员,从不同的地点,以不同的接入方式(比如 LAN, WAN, Internet/Intranet 等)访问和操作共同的数据;最大的缺点是对外网环境依赖性太强,由于各种原因引起企业外网中断都会造成系统瘫痪。B-S 编程模式,主要是利用了不断成熟的 WWW 浏览器技术,结合浏览器的多种 Script 语言(VBScript、JavaScript 等)和 ActiveX 技术,用通用浏览器就实现了原来需要复杂专用软件才能实现的强大功能,并节约了开发成本。

11.1.4 Socket 网络编程接口

套接字(Socket)即应用进程和网络协议间的接口,通过套接字可以很方便地实现应用进程间的通信。

1. Socket 简介

Socket 编程接口的发展历史可追溯至 20 世纪 80 年代初,当时美国国防高级研究计划局(Advanced Research Projects Agency, ARPA)为了资助加利福尼亚大学伯克利分校的一

个研究组,将 TCP/IP 移植到了 UNIX 系统中,同时 Sun 公司的创始人之一兼首席技术官 Bill Joy(比尔·乔伊)于 1977 年在伯克利分校开发了 TCP/IP 应用程序接口,即 BSD Socket 接口,TCP/IP 应用进程使用这个接口可以方便地进行通信。采用了 BSD Socket 接口的 UNIX 系统被称为 BSD UNIX(Berkeley Software Distribution UNIX)操作系统。由于许多计算机厂商都采用了 BSD UNIX,BSD Socket 接口得到了迅速普及并被广泛使用。

BSD Socket 共提供了三十多个接口函数,使用这些函数可以实现套接字的创建、地址绑定、连接建立、通信等功能。Go 语言标准库对这些函数进行了抽象和封装,在后续的举例中将逐渐介绍如何使用 Go 进行 Socket 网络编程。

2. Socket 类型

由于支持的底层协议不同,Socket 接口通常被分为三种类型:

(1) 流式套接字(SOCK_STREAM),这种套接字在 TCP 的基础上,提供了一个面向连接、可靠的双向数据流传输服务,数据流被无差错、无重复地发送,且按发送顺序接收。另外,使用流式套接字还可以实现带外数据传输服务。

(2) 数据报套接字(SOCK_DGRAM),这种套接字使用 UDP,提供一种高效的,但不保证可靠性、完整性和顺序性的双向数据报传输服务。使用数据报套接字还可以实现一些特殊通信服务,比如广播通信。

(3) 原始式套接字(SOCK_RAW)。这种套接字在 IP 的基础上,允许进程直接对较低层协议如 ICMP、IGMP 进行访问,如 Ping 程序就是使用原始套接字设计的。用户还可以在原始套接字的基础上,实现高层自定义协议。

11.2 Go 网络编程基础

Go 语言关于网络编程的所有数据结构、函数和方法都定义在 net 包里,所以要学习 Go 网络编程技术,首先就要熟悉 net 包。本节主要结合 net 包,介绍一些网络编程的基础知识。

11.2.1 IP 地址和域名解析

主机地址是网络通信最重要的数据之一,net 包中定义了三种类型的主机地址数据类型:IP、IPMask 和 IPAddr,它们分别用来存储协议相关的网络地址。

1. IP 地址类型

在 net 包中,IP 地址类型被定义成一个 byte 型数组,即若干个 8 位组,格式如下:

```
type IP []byte
```

在 net 包中,有几个函数可以将 IP 地址类型作为函数的返回类型,比如 ParseIP() 函数,该函数原型定义如下:

```
func ParseIP(s string) IP
```

ParseIP()函数的主要作用是分析IP地址的合法性,如果是一个合法的IP地址,ParseIP()函数将返回一个IP地址对象。如果是一个非法IP地址,ParseIP()函数将返回nil。

还可以使用IP对象的String()方法将IP地址转换成字符串格式,String()方法的原型定义如下:

```
func (ip IP) String() string
```

如果是IPv4地址,String()方法将返回一个点分十进制格式的IP地址,如“192.168.0.1”。如果是IPv6地址,String()方法将返回使用“:”分隔的地址形式,如“2000:0:0:0:0:0:0:1”。另外注意一个特例,对于地址“0:0:0:0:0:0:0:1”的返回结果是省略格式“::1”。

例 11-1 IP 地址类型。

```
1 //IP 地址类型
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     if len(os.Args) != 2 {
11         fmt.Fprintf(os.Stderr, "Usage: %s ip.addr\n", os.Args[0])
12         os.Exit(1)
13     }
14     addr := os.Args[1]
15     ip := net.ParseIP(addr)
16     if ip == nil {
17         fmt.Println("Invalid address")
18     } else {
19         fmt.Println("The address is", ip.String())
20     }
21     os.Exit(0)
22 }
```

编译并运行该程序,测试过程如下:

```
从键盘输入: ip 192.168.0.1 ✓
输出结果为: The address is 192.168.0.1
从键盘输入: ip 192.168.0.256 ✓
输出结果为: Invalid address
从键盘输入: ip 0:0:0:0:0:0:0:1 ✓
输出结果为: ::1
```

2. IPMask 地址类型

在Go语言中,为了方便子网掩码操作与计算,net包中还提供了IPMask地址类型。

在前面讲过,子网掩码地址其实就是一个特殊的 IP 地址,所以 IPMask 类型也是一个 byte 型数组,格式如下:

```
type IPMask []byte
```

函数 IPv4Mask() 可以通过一个 32 位 IPv4 地址生成子网掩码地址,调用成功后返回一个 4 字节的十六进制子网掩码地址。IPv4Mask() 函数原型定义如下:

```
func IPv4Mask(a, b, c, d byte) IPMask
```

另外,还可以使用主机地址对象的 DefaultMask() 方法获取主机默认子网掩码地址,DefaultMask() 方法原型定义如下:

```
func (ip IP) DefaultMask() IPMask
```

要注意的是,只有 IPv4 地址才有默认子网掩码。如果不是 IPv4 地址,DefaultMask() 方法将返回 nil。不管是通过调用 IPv4Mask() 函数,还是执行 DefaultMask() 方法,获取的子网掩码地址都是十六进制格式的。例如,子网掩码地址“255.255.255.0”的十六进制格式是“ffffff00”。

主机地址对象还有一个 Mask() 方法,执行 Mask() 方法后,会返回 IP 地址与子网掩码地址相“与”的结果,这个结果即是主机所处的网络的“网络地址”。Mask() 方法原型定义如下:

```
func (ip IP) Mask(mask IPMask) IP
```

还可以通过子网掩码对象的 Size() 方法获取掩码位数(ones)和掩码总长度(bits),如果是一个非标准的子网掩码地址,则 Size() 方法将返回“0,0”。Size() 方法的原型定义如下:

```
func (m IPMask) Size() (ones, bits int)
```

例 11-2 子网掩码地址。

```
1 //子网掩码地址
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     if len(os.Args) != 2 {
11         fmt.Fprintf(os.Stderr, "Usage: %s ip.addr\n", os.Args[0])
12         os.Exit(1)
```



```

13     }
14     dotaddr := os.Args[1]
15     addr := net.ParseIP(dotaddr)
16     if addr == nil {
17         fmt.Println("Invalid address")
18     }
19     mask := addr.DefaultMask()
20     fmt.Println("Subnet mask is: " + mask.String())
21     network := addr.Mask(mask)
22     fmt.Println("Network address is:", network.String())
23     ones, bits := mask.Size()
24     fmt.Println("Mask bits:", ones, "Total bits:", bits)
25     os.Exit(0)
26 }

```

编译并运行该程序,测试过程如下:

```

从键盘输入: mask 192.168.0.1 ✓
输出结果为: Subnet mask is: ffffffff
              Network address is: 192.168.0.0
              Mask bits: 24 Total bits: 32

```

3. 域名解析

在 net 包中,许多函数或方法调用后返回的是一个指向 IPAddr 结构体的指针,结构体 IPAddr 内只定义了一个 IP 类型的字段,格式如下:

```

type IPAddr struct {
    IP IP
}

```

IPAddr 结构体的主要作用是用于域名解析服务(DNS),例如,函数 ResolveIPAddr() 可以通过主机名解析主机网络地址。ResolveIPAddr() 函数原型定义如下:

```

func ResolveIPAddr(net, addr string) (* IPAddr, error)

```

在调用 ResolveIPAddr() 函数时,参数 net 表示网络类型,可以是“ip”、“ip4”或“ip6”,参数 addr 可以是 IP 地址或域名,如果是 IPv6 地址则必须使用“[]”括起来。ResolveIPAddr() 函数调用成功后返回指向 IPAddr 结构体的指针,调用失败返回错误类型 error。

例 11-3 DNS 域名解析。

```

1 //DNS 域名解析
2 package main
3
4 import(
5     "fmt"
6     "net"

```

```

7     "os"
8 )
9 func main() {
10     if len(os.Args) != 2 {
11         fmt.Fprintf(os.Stderr, "Usage: %s hostname\n", os.Args[0])
12         fmt.Println("Usage: ", os.Args[0], "hostname")
13         os.Exit(1)
14     }
15     name := os.Args[1]
16     addr, err := net.ResolveIPAddr("ip", name)
17     if err != nil {
18         fmt.Println("Resolution error", err.Error())
19         os.Exit(1)
20     }
21     fmt.Println("Resolved address is ", addr.String())
22     os.Exit(0)
23 }

```

编译并运行该程序,测试过程如下:

从键盘输入: `ResolveIP www.google.com` ✓
 输出结果为: `Resolved address is 173.194.72.147`

11.2.2 主机信息查询

函数 `ResolveIPAddr()` 虽然可以利用主机名获取一个主机 IP 地址,但是大多数网络中的主机都拥有多个 IP 地址,因为在一块 NIC(网络接口设备)上往往能绑定多个 IP 地址。另外,一台主机也可以有多个名字,比如 `alias`(主机别名)。

1. 主机地址信息查询

可以使用 `LookupHost()` 函数查询主机更为详尽的信息,它能利用本地查询器获取主机信息,该函数原型定义如下:

```
func LookupHost(host string) (addrs []string, err error)
```

在调用 `LookupHost()` 函数时,参数 `host` 是字符串型的主机名,调用成功后函数返回数组格式的主机地址列表,否则返回一个错误类型。

例 11-4 获取主机地址信息。

```

1 //获取主机地址信息
2 package main
3
4 import(
5     "fmt"
6     "net"

```

```

7   "os"
8 )
9 func main() {
10    if len(os.Args) != 2 {
11        fmt.Fprintf(os.Stderr, "Usage: %s hostname\n", os.Args[0])
12        os.Exit(1)
13    }
14    name := os.Args[1]
15    addrs, err := net.LookupHost(name)
16    if err != nil {
17        fmt.Println("LookupHost error: ", err.Error())
18        os.Exit(1)
19    }
20    for _, s := range addrs {
21        fmt.Println(s)
22    }
23    os.Exit(0)
24 }

```

编译并运行该程序,测试过程如下:

从键盘输入: LookupHost www.baidu.com ✓

输出结果为: 119.75.218.77

119.75.217.56

119.75.217.56

注意,LookupHost()函数直接返回点分十进制格式的IP地址,而不是数值格式的IP地址,如果要对地址进行计算还要将其再转换为数值格式。

2. 主机正式名查询

在主机的多个名字信息中,有一个会被标记为“canonical”,即主机正式名。如果想查询主机正式名,可以使用函数LookupCNAME()来完成,该函数原型定义如下:

```
func LookupCNAME(name string) (cname string, err error)
```

例 11-5 获取主机正式名。

```

1 //获取主机正式名
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )

```



```
9 func main() {
10     if len(os.Args) != 2 {
11         fmt.Fprintf(os.Stderr, "Usage: %s hostname\n", os.Args[0])
12         os.Exit(1)
13     }
14     name := os.Args[1]
15     cname, err := net.LookupCNAME(name)
16     if err != nil {
17         fmt.Println("LookupCNAME error: ", err.Error())
18         os.Exit(1)
19     }
20     fmt.Println("The host canonical name is: ", cname)
21     os.Exit(0)
22 }
```

编译并运行该程序,测试过程如下:

从键盘输入: LookupCNAME www.sina.com.cn ✓

输出结果为: The host canonical name is: jupiter.sina.com.cn.

11.2.3 服务信息查询

在 C-S 编程模式中,相同的服务进程会驻留在不同的主机系统中,也就是说同一种服务可以由不同的主机来提供,比如 Internet 中有很多 Web 服务器。由于每一台主机都有自己的全球唯一的 IP 地址,所以可以通过 IP 地址来区分相同的服务到底是由哪一台主机提供的。

1. SRV 记录查询

SRV 是 DNS 服务器的数据库中支持的一种资源记录的类型,它记录了哪台计算机提供了哪个服务这么一个简单的信息。

SRV 记录:一般是为 Microsoft 的活动目录设置时的应用。DNS 可以独立于活动目录,但是活动目录必须有 DNS 的帮助才能工作。为了活动目录能够正常工作,DNS 服务器必须支持服务定位(SRV)资源记录,资源记录把服务名字映射为提供服务的服务器名字。活动目录客户和域控制器使用 SRV 资源记录决定域控制器的 IP 地址。

函数 LookupSRV() 可以用来查询 SRV 信息,该函数原型定义如下:

```
func LookupSRV(service, proto, name string) (cname string, addrs [] * SRV, err error)
```

在调用函数 LookupSRV() 时,参数 service 表示服务名,参数 proto 表示通信协议(TCP 或 UDP),参数 name 是主机域名。函数调用成功会返回主机正式名和 SRV 地址列表,否则返回错误类型。

例 11-6 查询服务器。

```
1 //查询服务器
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     if len(os.Args) != 4 {
11         fmt.Fprintf(os.Stderr, "Usage: % sservice proto name\n", os.Args[0])
12         os.Exit(1)
13     }
14     service := os.Args[1]
15     proto := os.Args[2]
16     name := os.Args[3]
17     cname, addrs, err := net.LookupSRV(service, proto, name)
18     if err != nil {
19         fmt.Println("LookupSRV error: ", err.Error())
20         os.Exit(1)
21     }
22     fmt.Println("The service canonical name is: ", cname)
23     for _, addr := range addrs {
24         fmt.Println("The service addr list is:", addr.Target)
25     }
26     os.Exit(0)
27 }
```

编译并运行该程序,测试过程如下:

从键盘输入: LookupSRV xmpp - server tcp gocn. im ✓

输出结果为: The service canonical name is: gocn. im

The service addr list is: xmpp - server. l. google. com

2. 服务端口查询

同样地,在一台主机上也可以安装多个服务进程,这些服务进程分别提供不同的网络服务,比如 Web 服务、FTP 服务、DNS 服务等。通过前面的知识知道,可以利用服务端口号来区分不同的网络服务。

函数 LookupPort() 可以用来查询服务端口号,该函数原型定义如下:

```
func LookupPort(network, service string) (port int, err error)
```

在调用函数 `LookupPort()` 时,参数 `network` 是网络类型,参数 `service` 是服务类型,比如 WWW 服务、FTP 服务、Telnet 服务等。函数调用成功会返回服务端口号,否则返回一个错误类型。

例 11-7 查询服务端口号。

```
1 //查询服务端口号
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     if len(os.Args) != 3 {
11         fmt.Fprintf(os.Stderr, "Usage: %s networkType service\n", os.Args[0])
12         os.Exit(1)
13     }
14     networkType := os.Args[1]
15     service := os.Args[2]
16     port, err := net.LookupPort(networkType, service)
17     if err != nil {
18         fmt.Println("LookupPort error: ", err.Error())
19         os.Exit(1)
20     }
21     fmt.Println("The service port is: ", port)
22     os.Exit(0)
23 }
```

编译并运行该程序,测试过程如下:

从键盘输入: `LookupPORT tcp www` ✓

输出结果为: `The service port is: 80`

它会随时准备接收客户机的连接请求并做出响应。服务器有多种类型,它们会通过不同的方式为客户机提供服务。在 Internet 中,大部分服务器都是通过 TCP 或 UDP 进行通信,尽管也有一些替代协议(比如 SCTP)可用。有许多应用层的协议都是基于 TCP、UDP 实现的,比如 P2P(Peer-to-Peer),远程过程调用(RPC),通信代理(Communicating Agents)等。

11.3 Go 网络编程原理

使用 Go 语言编写网络程序时,它和传统的 BSD Socket 或 WinSock 网络编程形式不一样。

11.3.1 Socket 网络编程

传统的 Socket 网络编程分为：流式套接字(SOCK_STREAM, 基于 TCP)、数据报套接字(SOCK_DGRAM, 基于 UDP)、原始式套接字(SOCK_RAW, 基于 IP)几大类。比如流式套接字编程会按如下步骤进行：

- (1) 服务器、客户机分别调用 `socket()` 函数创建套接字。
- (2) 服务器、客户机分别调用 `bind()` 函数绑定服务器地址。
- (3) 服务器调用 `listen()` 进行监听, 而客户机会调用 `connect()` 函数向服务器发出连接请求。

(4) 服务器调用 `accept()` 函数接受客户机的连接请求, 并返回一个新的套接字用于处理和客户机之间的通信连接。

(5) 经过前面几个步骤, 服务器和客户机之间的通信链路已经建立成功, 这时就可以调用 `send()` 或 `recv()` 函数互发数据或接收数据了。

(6) 如果通信结束, 任何一方都可以调用 `close()` 函数关闭套接字, 断开通信连接。

而在使用数据报套接字编程时, 则不需要监听、连接等步骤, 服务器、客户机只要建立好套接字后就可以直接互发数据, 通信过程更简单。

11.3.2 Go 网络编程

Go 语言标准库对传统 Socket 网络编程的过程进行了封装, 无论期望使用什么协议建立什么形式的连接, 都只需要调用 `Dial()` 函数即可。 `Dial()` 函数的原型定义如下：

```
func Dial(net, addr string) (Conn, error)
```

在 `Dial()` 函数中, 参数 `net` 是网络协议名, 参数 `addr` 是 IP 地址或域名, 而端口号以“:”的形式跟随在地址或域名的后面, 端口是可选的。 `Dial()` 函数如果调用成功, 则返回一个连接对象。否则, 返回一个错误类型。

例如, 建立一个 TCP 连接, 主机地址为“192.168.0.1”, 端口号为 5000, 代码如下：

```
conn, err := net.Dial("tcp", "192.168.0.1:5000")
```

例如, 建立一个 UDP 连接, 主机地址为“192.168.0.2”, 端口号为 5001, 代码如下：

```
conn, err := net.Dial("udp", "192.168.0.2:5001")
```

例如, 建立一个 ICMP 连接, 使用协议名, 主机域名为 `www.baidu.com`, 省略端口号, 代码如下：

```
conn, err := net.Dial("ip4:icmp", "www.baidu.com")
```

例如, 建立一个 ICMP 连接, 使用协议号, 主机地址为“119.75.218.77”, 省略端口号, 代码如下：

```
conn, err := net.Dial("ip4:1", "119.75.218.77")
```

目前, Dial() 函数支持如下几种网络协议: “tcp”、“tcp4”(仅限 IPv4)、“tcp6”(仅限 IPv6)、“udp”、“udp4”(仅限 IPv4)、“udp6”(仅限 IPv6)、“ip”、“ip4”(仅限 IPv4)、“ip6”(仅限 IPv6)。

在成功建立连接后, 就可以进行数据的发送和接收了。发送数据时, 可以使用 conn 对象的 Write() 成员方法, 在接收数据时, 可以使用 conn 对象的 Read() 成员方法。

11.4 TCP 网络程序设计

TCP 工作在网络的传输层, 它属于一种面向连接的可靠的通信协议。TCP 网络程序设计属于 C-S 模式, 一般要设计一个服务器程序, 一个或多个客户机程序。另外, TCP 是面向连接的通信协议, 所以客户机要和服务器进行通信, 首先要在通信双方之间建立通信连接。本节将详细讲解 TCP 网络编程服务器、客户机的设计原理和设计过程。

11.4.1 TCPAddr 地址结构体

在进行 TCP 网络编程时, 服务器或客户机的地址使用 TCPAddr 地址结构体表示, TCPAddr 包含两个字段: IP 和 Port, 形式如下:

```
type TCPAddr struct {
    IP IP
    Port int
}
```

函数 ResolveTCPAddr() 可以把网络地址转换为 TCPAddr 地址结构, 该函数原型定义如下:

```
func ResolveTCPAddr(net, addr string) (*TCPAddr, error)
```

在调用函数 ResolveTCPAddr() 时, 参数 net 是网络协议名, 可以是 “tcp”、“tcp4” 或 “tcp6”。参数 addr 是 IP 地址或域名, 如果是 IPv6 地址则必须使用 “[]” 括起来。另外, 端口号以 “:” 的形式跟随在 IP 地址或域名的后面, 端口是可选的。例如: “www.google.com:80” 或 “127.0.0.1:21”。还有一种特例, 就是对于 HTTP 服务器, 当主机地址为本地测试地址时 (127.0.0.1), 可以直接使用端口号作为 TCP 连接地址, 形如 “:80”。

函数 ResolveTCPAddr() 调用成功后返回一个指向 TCPAddr 结构体的指针, 否则返回一个错误类型。

另外, TCPAddr 地址对象还有两个方法: Network() 和 String(), Network() 方法用于返回 TCPAddr 地址对象的网络协议名, 比如 “tcp”; String() 方法可以将 TCPAddr 地址转换成字符串形式。这两个方法原型定义如下:

```
func (a * TCPAddr) Network() string
func (a * TCPAddr) String() string
```

例 11-8 TCP 连接地址。

```
1 //TCP 连接地址
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     if len(os.Args) != 3 {
11         fmt.Fprintf(os.Stderr, "Usage: % snetworkType addr\n", os.Args[0])
12         os.Exit(1)
13     }
14     networkType := os.Args[1]
15     addr := os.Args[2]
16     tcpAddr, err := net.ResolveTCPAddr(networkType, addr)
17     if err != nil {
18         fmt.Println("ResolveTCPAddr error: ", err.Error())
19         os.Exit(1)
20     }
21     fmt.Println("The network type is: ", tcpAddr.Network())
22     fmt.Println("The IP address is", tcpAddr.String())
23     os.Exit(0)
24 }
```

编译并运行该程序,测试过程如下:

```
从键盘输入: ResolveTCPAddr tcp www.sohu.com:80 ✓
输出结果为: The network type is: tcp
              The IP address is 117.34.8.50:80
```

11.4.2 TCPConn 对象

在进行 TCP 网络编程时,客户机和服务器之间是通过 TCPConn 对象实现连接的, TCPConn 是 Conn 接口的实现。TCPConn 对象绑定了服务器的网络协议和地址信息, TCPConn 对象定义如下:

```
type TCPConn struct {
    //空结构
}
```

通过 TCPConn 连接对象,可以实现客户机和服务器间的全双工通信。可以通过 TCPConn 对象的 Read()方法和 Write()方法,在服务器和客户机之间发送和接收数据。Read()方法

和 Write() 方法的原型定义如下:

```
func (c * TCPConn) Read(b []byte) (n int, err error)
func (c * TCPConn) Write(b []byte) (n int, err error)
```

Read() 方法调用成功后会返回接收到的字节数, 调用失败返回一个错误类型; Write() 方法调用成功后会返回正确发送的字节数, 调用失败返回一个错误类型。另外, 这两个方法的执行都会引起阻塞。

11.4.3 TCP 服务器设计

前面讲了 Go 语言网络编程和传统 Socket 网络编程有所不同, TCP 服务器的工作过程如下:

(1) TCP 服务器首先注册一个公知端口, 然后调用 ListenTCP() 函数在这个端口上创建一个 TCPLListener 监听对象, 并在该对象上监听客户机的连接请求。

(2) 启用 TCPLListener 对象的 Accept() 方法接收客户机的连接请求, 并返回一个协议相关的 Conn 对象, 这里就是 TCPConn 对象。

(3) 如果返回了一个新的 TCPConn 对象, 服务器就可以调用该对象的 Read() 方法接收客户机发来的数据, 或者调用 Write() 方法向客户机发送数据了。

TCPLListener 对象、ListenTCP() 函数的原型定义如下:

```
type TCPLListener struct {
    //contains filtered or unexported fields
}
func ListenTCP(net string, laddr * TCPAddr) (* TCPLListener, error)
```

在调用函数 ListenTCP() 时, 参数 net 是网络协议名, 可以是“tcp”、“tcp4”或“tcp6”。参数 laddr 是服务器本地地址, 可以是任意活动的主机地址, 或者是内部测试地址“127.0.0.1”。该函数调用成功, 返回一个 TCPLListener 对象; 调用失败, 返回一个错误类型。

TCPLListener 对象的 Accept() 方法原型定义如下:

```
func (l * TCPLListener) Accept() (c Conn, err error)
```

Accept() 方法调用成功后, 返回 TCPConn 对象; 否则, 返回一个错误类型。

服务器和客户机的通信连接建立成功后, 就可以使用 Read() 和 Write() 方法收发数据。在通信过程中, 如果还想获取通信双方的地址信息, 可以使用 LocalAddr() 方法和 RemoteAddr() 方法来完成, 这两个方法原型定义如下:

```
func (c * TCPConn) LocalAddr() Addr
func (c * TCPConn) RemoteAddr() Addr
```

LocalAddr() 方法会返回本地主机地址, RemoteAddr() 方法返回远端主机地址。

例 11-9 TCP Server 端设计, 服务器使用本地地址, 服务端口号为 5000。服务器设计

工作模式采用循环服务器,对每一个连接请求调用线程 handleClient 来处理。

```
1 //TCP Server 端设计
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     service := ":5000"
11     tcpAddr, err := net.ResolveTCPAddr("tcp", service)
12     checkError(err)
13     listener, err := net.ListenTCP("tcp", tcpAddr)
14     checkError(err)
15     for {
16         conn, err := listener.Accept()
17         if err != nil {
18             continue
19         }
20         handleClient(conn)
21         conn.Close()
22     }
23 }
24 func handleClient(conn net.Conn) {
25     var buf [512]byte
26     for {
27         n, err := conn.Read(buf[0:])
28         if err != nil {
29             return
30         }
31         rAddr := conn.RemoteAddr()
32         fmt.Println("Receive from client", rAddr.String(), string(buf[0:n]))
33         _, err2 := conn.Write([]byte("Welcome client!"))
34         if err2 != nil {
35             return
36         }
37     }
38 }
39 func checkError(err error) {
40     if err != nil {
41         fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
42         os.Exit(1)
43     }
44 }
```


11.4.4 TCP 客户端设计

在 TCP 网络编程中,客户机的工作过程如下:

(1) TCP 客户机在获取了服务器的服务端口号和服务地址之后,可以调用 DialTCP() 函数向服务器发出连接请求,如果请求成功会返回 TCPConn 对象。

(2) 客户机调用 TCPConn 对象的 Read()或 Write()方法,与服务器进行通信活动。

(3) 通信完成后,客户机调用 Close()方法关闭连接,断开通信链路。

DialTCP()函数原型定义如下:

```
Func DialTCP(net string, laddr, raddr * TCPAddr) (* TCPConn, error)
```

在调用函数 DialTCP()时,参数 net 是网络协议名,可以是“tcp”、“tcp4”或“tcp6”。参数 laddr 是本地主机地址,可以设为 nil。参数 raddr 是对方主机地址,必须指定不能省略。函数调用成功后,返回 TCPConn 对象;调用失败,返回一个错误类型。

方法 Close()的原型定义如下:

```
func (c * TCPConn) Close() error
```

该方法调用成功后,关闭 TCPConn 连接;调用失败,返回一个错误类型。

例 11-10 TCP Client 端设计,客户机通过内部测试地址“127.0.0.1”和端口 5000 和服务器建立通信连接。

```
1 //TCP Client 端设计
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     var buf [512]byte
11     if len(os.Args) != 2 {
12         fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
13     }
14     service := os.Args[1]
15     tcpAddr, err := net.ResolveTCPAddr("tcp", service)
16     checkError(err)
17     conn, err := net.DialTCP("tcp", nil, tcpAddr)
18     checkError(err)
19     rAddr := conn.RemoteAddr()
20     n, err := conn.Write([]byte("Hello server!"))
21     checkError(err)
22     n, err = conn.Read(buf[0:])
23     checkError(err)
```



```
24     fmt.Println("Reply form server ", rAddr.String(), string(buf[0:n]))
25     conn.Close()
26     os.Exit(0)
27 }
28 func checkError(err error) {
29     if err != nil {
30         fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
31         os.Exit(1)
32     }
33 }
```

编译并运行服务器端和客户端,测试过程如下:

启动服务器: TCPServer ✓

客户机连接: TCPClient 127.0.0.1:5000 ✓

服务器响应: Receive from client 127.0.0.1:2800 Hello server!

客户机接收: Reply form server 127.0.0.1:5000 Welcome client!

从上述测试结果可以看出,服务器注册了一个公知端口 5000,而当客户机与服务器建立连接后,客户机会生成一个临时端口“2800”与服务器进行通信。服务器不管启动多少次端口号都是 5000,而客户机每一次重新启动端口号都不一样。

11.4.5 使用 Goroutine 实现并发服务器

在 11.4.3 节中服务器设计采用循环服务器设计模式,这种服务器设计简单但缺陷明显。因为这种服务器一旦启动,就一直阻塞监听客户机的连接请求,直至服务器关闭。所以,循环服务器很耗费系统资源。

解决问题的方法是采用并发服务器模式,在这种模式中,对每一个客户机的连接请求,服务器都会创建一个新的进程、线程或者协程进行响应,而服务器还可以去处理其他任务。在第 10 章已经了解到,Goroutine 即协程是一种比线程更轻量级的任务单位,所以这里就使用 Goroutine 来实现并发服务器的设计。

例 11-11 并发服务器设计,服务器使用本地地址,服务端口号为 5000。服务器设计工作模式采用并发服务器模式,对每一个连接请求创建一个能调用 `handleClient()` 函数的 Goroutine 来处理。

```
1 //TCP Server 端设计
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     service := ":5000"
```

```

11     tcpAddr, err := net.ResolveTCPAddr("tcp", service)
12     checkError(err)
13     listener, err := net.ListenTCP("tcp", tcpAddr)
14     checkError(err)
15     for {
16         conn, err := listener.Accept()
17         if err != nil {
18             continue
19         }
20         go handleClient(conn)           //创建 Goroutine
21     }
22 }
23 func handleClient(conn net.Conn) {
24     defer conn.Close()                 //逆序调用 Close()保证连接能正常关闭
25     var buf [512]byte
26     for {
27         n, err := conn.Read(buf[0:])
28         if err != nil {
29             return
30         }
31         rAddr := conn.RemoteAddr()
32         fmt.Println("Receive from client", rAddr.String(), string(buf[0:n]))
33         _, err2 := conn.Write([]byte("Welcome client!"))
34         if err2 != nil {
35             return
36         }
37     }
38 }
39 func checkError(err error) {
40     if err != nil {
41         fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
42         os.Exit(1)
43     }
44 }

```

编译并运行服务器端和客户端,测试过程如下:

启动服务器: TCPServer ✓

客户机连接: TCPClient 127.0.0.1:5000 ✓

服务器响应: Receive from client 127.0.0.1:4024 Hello server!

客户机接收: Reply form server 127.0.0.1:5000 Welcome client!

通过测试可以发现,并发服务器可以同时响应多个客户机的连接请求,并能和多个客户机并发通信,尤其在多核心系统平台上,这种通信模式效率更高。而循环服务器只能按客户机的请求队列次序,一个一个地为客户机提供通信服务,通信效率低下。

11.5 UDP 网络程序设计

UDP 和 TCP 一样,也工作在网络传输层,但和 TCP 不同的是,它提供不可靠的通信服务。UDP 网络编程也为 C-S 模式,要设计一个服务器,一个或多个客户机。另外,UDP 是不保证可靠性的通信协议,所以客户机和服务器之间只要建立连接,就可以直接通信,而不用调用 `Accept()` 进行连接确认。本节将详细讲解 UDP 网络编程服务器、客户机的设计原理和设计过程。

11.5.1 UDPAddr 地址结构体

在进行 UDP 网络编程时,服务器或客户机的地址使用 `UDPAddr` 地址结构体表示,`UDPAddr` 包含两个字段: `IP` 和 `Port`,形式如下:

```
type UDPAddr struct {  
    IP IP  
    Port int  
}
```

函数 `ResolveUDPAddr()` 可以把网络地址转换为 `UDPAddr` 地址结构,该函数原型定义如下:

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, error)
```

在调用函数 `ResolveUDPAddr()` 时,参数 `net` 是网络协议名,可以是“udp”、“udp4”或“udp6”。参数 `addr` 是 IP 地址或域名,如果是 IPv6 地址则必须使用“`[]`”括起来。另外,端口号以“:”的形式跟随在 IP 地址或域名的后面,端口是可选的。

函数 `ResolveUDPAddr()` 调用成功后返回一个指向 `UDPAddr` 结构体的指针,否则返回一个错误类型。

另外,`UDPAddr` 地址对象还有两个方法: `Network()` 和 `String()`,`Network()` 方法用于返回 `UDPAddr` 地址对象的网络协议名,比如“udp”;`String()` 方法可以将 `UDPAddr` 地址转换成字符串形式。这两个方法原型定义如下:

```
func (a *UDPAddr) Network() string  
func (a *UDPAddr) String() string
```

11.5.2 UDPCConn 对象

在进行 UDP 网络编程时,客户机和服务器之间是通过 `UDPCConn` 对象实现连接的,`UDPCConn` 是 `Conn` 接口的实现。`UDPCConn` 对象绑定了服务器的网络协议和地址信息。`UDPCConn` 对象定义如下:


```
type UDPConn struct {
    //空结构
}
```

通过 UDPConn 连接对象在客户机和服务器之间进行通信,UDP 并不能保证通信的可靠性和有序性,这些都要由程序员来处理。为此,TCPConn 对象提供了 ReadFromUDP() 方法和 WriteToUDP() 方法,这两个方法直接使用远端主机地址进行数据发送和接收,即便在链路失效的情况下,通信操作都能正常进行。ReadFromUDP() 方法和 WriteToUDP() 方法的原型定义如下:

```
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err error)
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
```

ReadFromUDP() 方法调用成功后返回接收字节数和发送方地址,否则返回一个错误类型; WriteToUDP() 方法调用成功后返回发送字节数;否则返回一个错误类型。

11.5.3 UDP 服务器设计

在 UDP 网络编程中,服务器工作过程如下:

(1) UDP 服务器首先注册一个公知端口,然后调用 ListenUDP() 函数在这个端口上创建一个 UDPConn 连接对象,并在该对象上和客户机建立不可靠连接。

(2) 如果服务器和某个客户机建立了 UDPConn 连接,就可以使用该对象的 ReadFromUDP() 方法和 WriteToUDP() 方法相互通信了。

(3) 不管上一次通信是否完成或正常,UDP 服务器依然会接受下一次连接请求。

函数 ListenUDP() 原型定义如下:

```
func ListenUDP(net string, laddr *UDPAddr) (*UDPConn, error)
```

在调用函数 ListenUDP() 时,参数 net 是网络协议名,可以是“udp”、“udp4”或“udp6”。参数 laddr 是服务器本地地址,可以是任意活动的主机地址,或者是内部测试地址“127.0.0.1”。该函数调用成功,返回一个 UDPConn 对象;调用失败,返回一个错误类型。

例 11-12 UDP Server 端设计,服务器使用本地地址,服务端口号为 5001。服务器设计工作模式采用循环服务器,对每一个连接请求调用线程 handleClient 来处理。

```
1 //UDP Server 端设计
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     service := ":5001"
```

```

11     udpAddr, err := net.ResolveUDPAddr("udp", service)
12     checkError(err)
13     conn, err := net.ListenUDP("udp", udpAddr)
14     checkError(err)
15     for {
16         handleClient(conn)
17     }
18 }
19 func handleClient(conn * net.UDPConn) {
20     var buf [512]byte
21     n, addr, err := conn.ReadFromUDP(buf[0:])
22     if err != nil {
23         return
24     }
25     fmt.Println("Receive from client ", addr.String(), string(buf[0:n]))
26     conn.WriteToUDP([]byte("Welcome Client!"), addr)
27 }
28 func checkError(err error) {
29     if err != nil {
30         fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
31         os.Exit(1)
32     }
33 }

```

11.5.4 UDP 客户机设计

在 UDP 网络编程中,客户机工作过程如下:

(1) UDP 客户机在获取了服务器的服务端口号和服务地址之后,可以调用 DialUDP() 函数向服务器发出连接请求,如果请求成功会返回 UDPConn 对象。

(2) 客户机可以直接调用 UDPConn 对象的 ReadFromUDP() 方法或 WriteToUDP() 方法,与服务器进行通信活动。

(3) 通信完成后,客户机调用 Close() 方法关闭 UDPConn 连接,断开通信链路。

函数 DialUDP() 原型定义如下:

```
func DialUDP(net string, laddr, raddr * UDPAddr) (* UDPConn, error)
```

在调用函数 DialUDP() 时,参数 net 是网络协议名,可以是“udp”、“udp4”或“udp6”。参数 laddr 是本地主机地址,可以设为 nil。参数 raddr 是对方主机地址,必须指定不能省略。函数调用成功后,返回 UDPConn 对象;调用失败,返回一个错误类型。

方法 Close() 的原型定义如下:

```
func (c * UDPConn) Close() error
```

该方法调用成功后,关闭 UDPConn 连接;调用失败,返回一个错误类型。

例 11-13 UDP Client 端设计, 客户机通过内部测试地址“127.0.0.1”和端口 5001 和服务器建立通信连接。

```

1 //UDP Client 端设计
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     if len(os.Args) != 2 {
11         fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
12     }
13     service := os.Args[1]
14     udpAddr, err := net.ResolveUDPAddr("udp", service)
15     checkError(err)
16     conn, err := net.DialUDP("udp", nil, udpAddr)
17     checkError(err)
18     _, err = conn.Write([]byte("Hello Server!"))
19     checkError(err)
20     var buf [512]byte
21     n, addr, err := conn.ReadFromUDP(buf[0:])
22     checkError(err)
23     fmt.Println("Reply form server ", addr.String(), string(buf[0:n]))
24     conn.Close()
25     os.Exit(0)
26 }
27 func checkError(err error) {
28     if err != nil {
29         fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
30         os.Exit(1)
31     }
32 }

```

编译并运行服务器端和客户端, 测试过程如下:

启动服务器: UDPServer ✓

客户机连接: UDPCClient 127.0.0.1:5001 ✓

服务器响应: Receive from client 127.0.0.1:1662 Hello Server!

客户机接收: Reply form server 127.0.0.1:5001 Welcome Client!

通过测试结果会发现, 采用 TCP 时必须先启动服务器, 然后才能正常启动客户机, 如果服务器中断, 则客户机也会异常退出。而采用 UDP 时, 客户机和服务器启动没有先后次序, 而且即便是服务器异常退出, 客户机也能正常工作。

总之, TCP 可以保证客户机、服务器双方按照可靠有序的方式进行通信, 但通信效率低; 而 UDP 虽然不能保证通信的可靠性, 但通信效率要高得多, 在有些场合还是非常有用的。

11.6 IP 网络程序设计

IP 是 Internet 网络层的核心协议,它是一种不可靠的、无连接的通信协议。TCP、UDP 都是在 IP 的基础上实现的通信协议,所以 IP 属于一种底层协议,它可以直接对网络数据包(Package)进行处理。另外,通过 IP 用户还可以实现自己的网络服务协议。本节将详细讲解 IP 网络编程服务器、客户机的设计原理和设计过程。

11.6.1 IPAddr 地址结构体

在进行 IP 网络编程时,服务器或客户机的地址使用 IPAddr 地址结构体表示,IPAddr 结构体只有一个字段 IP,形式如下:

```
type IPAddr struct {  
    IP IP  
}
```

通过了解 IPAddr 地址结构可以发现,IP 网络编程属于一种底层网络程序设计,它可以直接对 IP 包进行处理,所以 IPAddr 地址中没有端口地址,这个和 TCPAddr 地址结构、UDPAddr 地址结构都不同,在应用时要特别注意。

函数 ResolveIPAddr() 可以把网络地址转换为 IPAddr 地址结构,该函数原型定义如下:

```
func ResolveIPAddr(net, addr string) (* IPAddr, error)
```

在调用 ResolveIPAddr() 函数时,参数 net 表示网络类型,可以是“ip”、“ip4”或“ip6”,参数 addr 是 IP 地址或域名,如果是 IPv6 地址则必须使用“[]”括起来。

函数 ResolveIPAddr() 调用成功后返回一个指向 IPAddr 结构体的指针,否则返回一个错误类型。

另外,IPAddr 地址对象还有两个方法: Network() 和 String()。Network() 方法用于返回 IPAddr 地址对象的网络协议名,比如“ip”; String() 方法可以将 IPAddr 地址转换成字符串形式。这两个方法原型定义如下:

```
func (a * IPAddr) Network() string  
func (a * IPAddr) String() string
```

11.6.2 IPConn 对象

在进行 IP 网络编程时,客户机和服务器之间是通过 IPConn 对象实现连接的,IPConn 是 Conn 接口的实现。IPConn 对象绑定了服务器的网络协议和地址信息,IPConn 对象定义如下:

```
type IPConn struct {
    //空结构
}
```

由于 IPConn 是一个无连接的通信对象,所以 IPConn 对象提供了 ReadFromIP() 方法和 WriteToIP() 方法用于在客户机和服务器之间进行数据收发操作。ReadFromIP() 和 WriteToIP() 的原型定义如下:

```
func (c * IPConn) ReadFromIP(b []byte) (int, * IPAddr, error)
func (c * IPConn) WriteToIP(b []byte, addr * IPAddr) (int, error)
```

ReadFromIP() 方法调用成功后返回接收字节数和发送方地址,否则返回一个错误类型; WriteToIP() 方法调用成功后返回发送字节数,否则返回一个错误类型。

11.6.3 IP 服务器设计

由于工作在网络层,IP 服务器并不需要在一个指定的端口上和客户机进行通信连接,IP 服务器的工作过程如下:

(1) IP 服务器使用指定的协议簇和协议,调用 ListenIP() 函数创建一个 IPConn 连接对象,并在该对象和客户机间建立不可靠连接。

(2) 如果服务器和某个客户机建立了 IPConn 连接,就可以使用该对象的 ReadFromIP() 方法和 WriteToIP() 方法相互通信了。

(3) 如果通信结束,服务器还可以调用 Close() 方法关闭 IPConn 连接。

函数 ListenIP() 原型定义如下:

```
func ListenIP(netProto string, laddr * IPAddr) (* IPConn, error)
```

在调用函数 ListenIP() 时,参数 netProto 是“网络类型+协议名”或“网络类型+协议号”,中间用“:”隔开,比如“IP4: IP”或“IP4: 4”。参数 laddr 是服务器本地地址,可以是任意活动的主机地址,或者是内部测试地址“127.0.0.1”。该函数调用成功,返回一个 IPConn 对象;调用失败,返回一个错误类型。

例 11-14 IP Server 端设计,服务器使用本地主机地址,调用 Hostname() 函数获取。服务器设计工作模式采用循环服务器,对每一个连接请求调用线程 handleClient 来处理。

```
1 //IP Server 端设计
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     name, err := os.Hostname()
```

```

11  checkError(err)
12  ipAddr, err := net.ResolveIPAddr("ip4", name)
13  checkError(err)
14  conn, err := net.ListenIP("ip4:ip", ipAddr)
15  checkError(err)
16  for {
17      handleClient(conn)
18  }
19 }
20 func handleClient(conn * net.IPConn) {
21     var buf [512]byte
22     n, addr, err := conn.ReadFromIP(buf[0:])
23     if err != nil {
24         return
25     }
26     fmt.Println("Receive from client ", addr.String(), string(buf[0:n]))
27     conn.WriteToIP([]byte("Welcome Client!"), addr)
28 }
29 func checkError(err error) {
30     if err != nil {
31         fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
32         os.Exit(1)
33     }
34 }

```

11.6.4 IP 客户机设计

在 IP 网络编程中,客户机工作过程如下:

(1) IP 客户机在获取了服务器的网络地址之后,可以调用 DialIP() 函数向服务器发出连接请求,如果请求成功会返回 IPConn 对象。

(2) 如果连接成功,客户机可以直接调用 IPConn 对象的 ReadFromIP() 方法或 WriteToIP() 方法,与服务器进行通信活动。

(3) 通信完成后,客户机调用 Close() 方法关闭 IPConn 连接,断开通信链路。

函数 DialIP() 原型定义如下:

```
func DialIP(netProto string, laddr, raddr * IPAddr)( * IPConn, error)
```

在调用函数 DialIP() 时,参数 netProto 是“网络类型+协议名”或“网络类型+协议号”,中间用“:”隔开,比如“IP4: IP”或“IP4: 4”。参数 laddr 是本地主机地址,可以设为 nil。参数 raddr 是对方主机地址,必须指定不能省略。函数调用成功后,返回 IPConn 对象;调用失败,返回一个错误类型。

方法 Close() 的原型定义如下:

```
func (c * IPConn) Close() error
```


该方法调用成功后,关闭 IPConn 连接;调用失败,返回一个错误类型。

例 11-15 IP Client 端设计,客户机通过内部测试地址“127.0.0.1”和服务器建立通信连接,服务器主机地址可以使用 Hostname()函数获取。

```

1 //IP Client 端设计
2 package main
3
4 import(
5     "fmt"
6     "net"
7     "os"
8 )
9 func main() {
10     if len(os.Args) != 2 {
11         fmt.Fprintf(os.Stderr, "Usage: %s host", os.Args[0])
12     }
13     service := os.Args[1]
14     lAddr, err := net.ResolveIPAddr("ip4", service)
15     checkError(err)
16     name, err := os.Hostname()
17     checkError(err)
18     rAddr, err := net.ResolveIPAddr("ip4", name)
19     checkError(err)
20     conn, err := net.DialIP("ip4:ip", lAddr, rAddr)
21     checkError(err)
22     _, err = conn.WriteToIP([]byte("Hello Server!"), rAddr)
23     checkError(err)
24     var buf [512]byte
25     n, addr, err := conn.ReadFromIP(buf[0:])
26     checkError(err)
27     fmt.Println("Reply form server ", addr.String(), string(buf[0:n]))
28     conn.Close()
29     os.Exit(0)
30 }
31 func checkError(err error) {
32     if err != nil {
33         fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
34         os.Exit(1)
35     }
36 }

```

编译并运行服务器端和客户端,测试过程如下:

启动服务器: IPServer ✓

客户机连接: IPClient 127.0.0.1 ✓

服务器响应: Receive from client 127.0.0.1 Hello Server!

客户机接收: Reply form server 192.168.1.104 Welcome Client!

通过测试结果可以看出, TCP、UDP 的服务器和客户机通信时必须使用端口号,而 IP

服务器和客户机之间通信不需要端口号。另外,如果在同一台计算机上,服务器、客户机要使用不同的地址进行测试,比如本例服务器地址是“192.168.1.104”,客户机使用内部测试地址“127.0.0.1”。如果使用相同的地址,会发生自发自收的现象,原因是 IP 是底层通信,并没有像 TCP、UDP 那样使用端口号来区分不同的通信进程。

11.6.5 Ping 程序设计

不管是 UNIX 还是 Windows 系统中都有一个 Ping 命令,利用它可以检查网络是否连通,分析判断网络故障。Ping 会向目标主机发送测试数据包,看对方是否有响应并统计响应时间,以此测试网络。

Ping 命令的这些功能是使用 IP 层的 ICMP 实现的,在测试过程中,源主机向目标主机发送回显请求报文(ICMP_ECHO_REQUEST, type=8, code=0),目的主机返回回显响应报文(ICMP_ECHO_REPLY, type=0, code=0),相关的数据包格式如图 11-6 所示。其中,标识符是源主机的进程号,序列码用来标识发出回显请求的次序,时间戳表示数据包发出的时刻,通过比较回显响应时刻和源主机当前时刻的差值,可以测出 ICMP 数据包的往返时间。

类型(8/0)	代码(0)	校验和
标识符		序列码
时间戳(tv_sec)		
时间戳(tv_usec)		

图 11-6 ICMP 回显请求和响应数据包格式

例 11-16 使用原始套接字和 ICMP 设计 Ping 程序,函数 makePingRequest()的功能是生成 ICMP 请求包,函数 parsePingReply()用于解析目标主机发回的响应包,函数 elapsedTime()的功能是计算 ICMP 数据包往返时间。

```
1 //Ping 程序
2 package main
3
4 import(
5     "bytes"
6     "fmt"
7     "net"
8     "os"
9     "time"
10 )
11 const (
12     ICMP_ECHO_REQUEST = 8
13     ICMP_ECHO_REPLY  = 0
14 )
15 func main() {
16     if len(os.Args) != 2 {
```



```

17     fmt.Fprintf(os.Stderr, "Usage: %s host", os.Args[0])
18     os.Exit(1)
19 }
20 dst := os.Args[1]
21 raddr, err := net.ResolveIPAddr("ip4", dst)
22 checkError(err)
23 ipconn, err := net.DialIP("ip4:icmp", nil, raddr)
24 checkError(err)
25 sendid := os.Getpid() & 0xffff
26 sendseq := 1
27 pingpktlen := 64
28 for {
29     sendpkt := makePingRequest(sendid, sendseq, pingpktlen, []byte(""))
30     start := int64(time.Now().Nanosecond())
31     _, err := ipconn.WriteToIP(sendpkt, raddr)
32     checkError(err)
33     resp := make([]byte, 1024)
34     for {
35         n, from, err := ipconn.ReadFrom(resp)
36         checkError(err)
37         fmt.Printf("%d bytes from %s: icmp_req = %d time = %.2f ms\n", n,
38             from, sendseq, elapsedTime(start))
39         if resp[0] != ICMP_ECHO_REPLY {
40             continue
41         }
42         rcvvid, rcvseq := parsePingReply(resp)
43         if rcvvid != sendid || rcvseq != sendseq {
44             fmt.Println("Ping reply saw id ", rcvvid, rcvseq, sendid,
45                 sendseq)
46             break
47         }
48         if sendseq == 4 {
49             break
50         } else {
51             sendseq++
52         }
53         time.Sleep(1e9)
54     }
55 }
56 func makePingRequest(id, seq, pktlen int, filler []byte) []byte {
57     p := make([]byte, pktlen)
58     copy(p[8:], bytes.Repeat(filler, (pktlen-8)/len(filler)+1))
59     p[0] = ICMP_ECHO_REQUEST //type
60     p[1] = 0                 //code
61     p[2] = 0                 //cksum
62     p[3] = 0                 //cksum
63     p[4] = uint8(id >> 8)   //id
64     p[5] = uint8(id & 0xff) //id
65     p[6] = uint8(seq >> 8)  //sequence

```



```

65     p[7] = uint8(seq & 0xff) //sequence
66     cklen := len(p)
67     s := uint32(0)
68     for i := 0; i < (cklen-1); i+=2 {
69         s += uint32(p[i+1])<<8 | uint32(p[i])
70     }
71     if cklen&1 == 1 {
72         s += uint32(p[cklen-1])
73     }
74     s = (s >> 16) + (s & 0xffff)
75     s = s + (s >> 16)
76     p[2] ^= uint8(^s & 0xff)
77     p[3] ^= uint8(^s >> 8)
78     return p
79 }
80 func parsePingReply(p []byte) (id, seq int) {
81     id = int(p[4])<<8 | int(p[5])
82     seq = int(p[6])<<8 | int(p[7])
83     return
84 }
85 func elapsedTime(start int64) float32 {
86     t := float32((int64(time.Now().Nanosecond()) - start) / 1000000.0)
87     return t
88 }
89 func checkError(err error) {
90     if err != nil {
91         fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
92         os.Exit(1)
93     }
94 }

```

编译并运行 Ping 程序,测试结果如下:

```

Ping www.sina.com.cn ✓
64 bytes from 218.30.66.101: icmp_req=1 time=15 ms
64 bytes from 218.30.66.101: icmp_req=2 time=12 ms
64 bytes from 218.30.66.101: icmp_req=3 time=15 ms
64 bytes from 218.30.66.101: icmp_req=4 time=15 ms
Ping www.google.com.hk ✓
64 bytes from 125.76.239.243: icmp_req=1 time=31 ms
64 bytes from 125.76.239.243: icmp_req=2 time=15 ms
64 bytes from 125.76.239.243: icmp_req=3 time=23 ms
64 bytes from 125.76.239.243: icmp_req=4 time=15 ms

```

小结

本章主要介绍了 Go 语言中的网络编程技术,包括网络编程的基本概念和编程模式。Go 语言网络编程接口总体上来说还是继承了 UNIX 套接字网络编程接口,分为 TCP 网络编程接口、UDP 网络编程接口和 IP 网络编程接口。

通过这一章的学习,首先要掌握网络编程的基础知识,包括网络编程接口、网络编程模式、网络地址和主机信息查询等。在掌握上述基础知识的前提下,就可以使用 TCP、UDP 接口开发可靠的或非可靠的网络应用程序了。也可以使用 IP 接口开发网络底层应用服务,比如本章最后一个例子,就是使用 IP 接口和 ICMP 设计了 Ping 工具。

习题

- 11.1 简述 IP、TCP 和 UDP 的功能与作用。
- 11.2 IPv4 将 IP 地址划分为几种类型? 各有什么特点? IPv6 地址长度为多少位? IPv4 地址如何表示成 IPv6 地址? 试举例说明。
- 11.3 常见的网络编程模式有哪几种? 各有什么特点?
- 11.4 服务器并发处理的解决方案基本分为两种: _____ 服务器与 _____ 服务器,这两种方案分别适合于不同的网络服务类型。
- 11.5 由于支持的底层协议不同,Socket 网络编程接口通常被分为三种类型: _____、_____、_____和 _____。
- 11.6 编写程序,计算主机子网掩码地址,可参考例 11-2。
- 11.7 编写程序,查询新浪网在本地域共布放了几台主机,主机地址分别是多少? 可参考例 11-4。
- 11.8 使用 TCP 设计一个可以互发短消息的服务器端和客户端程序。
- 11.9 使用 UDP 设计一个可以发送广播报文的服务器。
- 11.10 试用 IP 实现 11.8 题的功能。

Go 语言内置关键字

Go 语言内置关键字如附表 A-1 所示。

附表 A-1 Go 语言内置关键字

关 键 字	功 能	关 键 字	功 能
break	退出本层循环	if	条件判断语句
case	switch 分支语句	import	导入非 main 包
chan	定义通道	interface	定义接口
const	定义常量	map	定义字典
continue	结束本次循环	package	声明包
default	switch 或 select 默认语句	range	遍历数组、切片或字典
defer	延迟退出	return	返回语句
else	if 分支语句	select	多通道或文件句柄监控机制
fallthrough	继续执行下一个 case 分支	struct	定义结构体
for	循环控制语句	switch	条件分支语句
func	定义函数或方法	type	自定义类型或接口
go	创建协程 goroutine	var	定义变量
goto	程序流程跳转语句		
reflect	运行时类型反射		
runtime	包含与 Go 运行时系统交互的函数		
sort	提供了对集合排序的函数		
sql	提供了一个通用的 SQL 接口		
strconv	提供了基本数据类型和字符串之间相互转换的一系列函数		
strings	提供了一系列对字符串进行操作的函数		
sync	提供了基本的同步机制，如互斥锁		
syscall	包含一个低级的操作系统原语接口		
testing	提供了对自动测试 Go 包的支持		
time	提供了一系列获取系统时间或显示时间的功能		
tar	实现了对 tar 压缩文件的访问		
unicode	提供了 Unicode 编码的相关函数		
zip	提供了对 ZIP 压缩文件的读和写的支持		

附录 B

Go 内置函数

- 11.1 简述 IP、TCP 和 UDP 的功能与作用。
- 11.2 IP 网将 IP 地址划分为几种类型？各有什么特点？IPv6 地址长度为多少位？
- Go 内置函数如附表 B-1 所示。
- 11.3 常见的网络编程模式有哪些？

附表 B-1 Go 内置函数

函 数 名	说 明
close	用于通道通信,关闭 channel
len	获取 string、array、slice、map、channel 的字节长度
cap	获取 slice 和 map 的存储容量
make	用于创建动态数据类型,比如 slice、map 和 channel,同时初始化内存容量并设置相关属性
new	通常用于值类型或用户自定义类型,比如 struct,为指定类型分配初始化过的存储空间,并返回指针
append	从 slice 的尾部开始,向其追加一个或多个元素
copy	在不同的 slice 之间复制数据
delete	从 map 中删除 key 对应的 value
complex	通过两个 float 数值,构建一个复数
real	返回复数 complex 的实部
imag	返回复数 complex 的虚部
panic	报告系统运行时错误,比如句柄操作错误,并停止常规的 goroutine 动作
recover	处理系统运行时错误,允许程序定义 goroutine 的 panic 动作

附录C

Go 语言标准库

Go 语言标准库如附表 C-1 所示。

附表 C-1 Go 语言标准库

包 名	说 明
bufio	通过 Reader、Writer 和 ReadWriter 对象实现了带缓冲的 I/O 接口
bytes	提供了大量对字节切片进行操作的函数
errors	通过 errors 接口实现了 Go 的错误机制
flag	实现了命令行标记解析
fmt	实现了格式化输入输出
hash	提供了哈希函数接口
html	实现了一个 HTML5 兼容的分词器和解析器
image	实现了一个基本的二维图像库
io	提供了对 I/O 接口的基本原语
math	提供了一些基本的数学常量和函数
net	提供了一个对 UNIX 网络套接字的可移植接口,包括 TCP/IP、UDP 域名解析和 UNIX 域套接字
os	为操作系统实现了一个平台无关的接口
path	实现了对斜线分割的文件名路径的操作
reflect	实行了运行时反射,允许一个程序以任意类型操作对象
runtime	包含与 Go 运行时系统交互的操作,如控制 goroutine 的函数
sort	提供了对集合排序的函数集
sql	提供了一个通用的 SQL 接口
strconv	提供了基本数据类型和字符串之间相互转换的一系列函数
strings	提供了一系列对字符串进行操作的函数
sync	提供了基本的同步机制,如互斥锁
syscall	包含一个低级的操作系统原语接口
testing	提供了对自动测试 Go 包的支持
time	提供了一系列获取系统时间或显示时间的功能
tar	实现了对 tar 压缩文件的访问
unicode	提供了 Unicode 编码的相关函数
zip	提供了对 ZIP 压缩文件的读和写的支持

名词与术语索引表

Go 内置函数如附表 B-1 所示。

. 示 1-C 表 附 词 语 术 语 言 语 Go

表 B-1 Go 内置函数

名 词	说 明	备 注
包(Package) 1.5.1	通过 Reader、Writer 和 ReadWriter 接口定义的函数。	bufio
闭包(Closure) 7.5.2	返回一个函数，该函数可以访问其定义所在函数的变量。	bytes
变参函数(Variable-argument Function) 7.4.1	通过 errors 包中的 Error 接口定义的函数。	errors
包导入(Import Package) 1.5.1	通过 fmt 包中的 Printf 函数定义的函数。	fmt
表达式(Expression) 2.3.1	通过 io 包中的 ReadWriter 接口定义的函数。	io
布尔型(Boolean) 2.2.1	通过 math 包中的 Basic 函数定义的函数。	math
并发(Concurrent) 10.1.1	通过 net 包中的 Listen 函数定义的函数。	net
并发服务器(Concurrent-server) 11.1.3	通过 os 包中的 ReadWriter 接口定义的函数。	os
包控制插件(Package Control) 1.4.2	通过 path 包中的 ReadWriter 接口定义的函数。	path
变量(Variable) 2.1.2	通过 reflect 包中的 ReadWriter 接口定义的函数。	reflect
变量名(Variable-name) 2.1.2	通过 runtime 包中的 ReadWriter 接口定义的函数。	runtime
变量值(Variable-value) 2.1.2	通过 sort 包中的 ReadWriter 接口定义的函数。	sort
别名(Alias) 2.2.5	通过 sql 包中的 ReadWriter 接口定义的函数。	sql
标签(Tag) 8.6.1	通过 strconv 包中的 ReadWriter 接口定义的函数。	strconv
标识符(Identifier) 2.1.3	通过 strings 包中的 ReadWriter 接口定义的函数。	strings
包声明(Package Declaration) 1.5.1	通过 sync 包中的 ReadWriter 接口定义的函数。	sync
C		
重复服务器(Interactive-server) 11.1.3	通过 syscall 包中的 ReadWriter 接口定义的函数。	syscall
超集(Superset) 9.3.1	通过 testing 包中的 ReadWriter 接口定义的函数。	testing
常量(Constant) 2.1.1	通过 time 包中的 ReadWriter 接口定义的函数。	time
Comma-ok 断言 9.4.1	通过 url 包中的 ReadWriter 接口定义的函数。	url
测试版(Test Version) 1.2.2	通过 unicode 包中的 ReadWriter 接口定义的函数。	unicode
参数传递(Parameter-passing) 7.3.1	通过 zip 包中的 ReadWriter 接口定义的函数。	zip
传输控制协议(Transmission Control Protocol, TCP) 11.1.1		
操作数(Operand) 2.3.1		
操作系统(Operating System, OS) 10.1.1		

D

- 对象(Object) 8.4.2
- 单分支选择结构(Single-branch Selection Structure) 4.1.1.1
- 多分支选择结构(Multi-branch Selection Structure) 4.1.1.1
- 递归调用(Recursibe-call) 7.6.1
- 递归函数(Recursibe-function) 7.6.1
- 端口号(Port Number) 11.1.2
- 多维数组(Multidimensional-array) 6.1.4
- 单向通道(Simplex-channel) 10.3.6

F

- 浮点型(Float) 2.2.3
- 方法(Method) 8.4.1
- 符号常量(Symbolic Constant) 2.1.1
- 复合语句(Compound Statement) 3.1.3
- 返回值(Return-value) 7.3.2
- 复数(Complex-number) 2.2.4
- 反射(Reflection) 9.5.1
- 服务器(Server) 11.1.3
- 赋值语句(Assignment Statement) 3.1.2

G

- Go 程(Goroutine) 10.2.2
- 共享内存(Shared Memory) 10.3.1
- 公知端口号(Well-known Port) 11.1.2
- 构造类型(Structure-type) 6.1.1
- 工作目录(Work Directory) 1.2.2

H

- 缓冲区(Buffer) 6.5.2
- 环境变量(Environment Variable) 1.2.2
- 函数(Function) 7.1.1
- 函数调用(Function-call) 7.2.1
- 函数声明(Function-declaration) 7.1.1

I

- IP 协议(Internet Protocol) 11.1.1

J

局部变量(Local Variables) 2.7.3

基本循环结构(For-loop Structure) 5.1.1

进程(Process) 10.2.1

进程号(Process ID,PID) 11.2.1

结构体(Strcut) 8.1.1

接口(Interface) 9.1.1

接口继承(Interface-inheritance) 9.1.3

接口组合(Interface-combination) 9.1.3

接收者(Receiver) 8.4.1

键值对(Key-value Pair) 6.3.1

K

空白标识(Blank-identifier) 7.3.2

客户端(Client) 11.1.3

开源(Open Source) 1.1.1

L

类(Class) 8.4.1

临时端口号(Ephemeral Port) 11.1.2

类型别名(Type-alias) 2.8.1

类型兼容性(Type-compatibility) 2.9.2

类型转换(Type Switch) 2.9.1

M

枚举类型(Enumeration-type) 2.6.1

面向对象(Object Oriented,OO) 8.4.2

内存地址(Memory Address) 2.2.7

H

N

匿名函数(Anonymous-function) 7.5.1

匿名字段(Anonymous-field) 8.3.1

内置关键字(Built-in Keyword) 2.1.3

内置函数(Built-in Function) 7.2.4

O

OSI/RM(Open System Interconnection/Reference Mode) 11.1.1

Q

全局变量(Global Variables) 2.7.3

嵌入式结构(Embedded Struct) 8.2.1

嵌套(Nested) 4.1.2

Y

S

Select 机制 10.4.1

双分支选择结构(Dual-branch Selection Structure) 4.1.1

实际参数(Actual Parameter) 7.3.1

数据管道(Pipe) 10.3.2

数据类型(Data-types) 2.2.1

数据输出(Data Output) 3.3.1

数据输入(Data Input) 3.3.2

顺序结构(Chronological Structure) 3.1.1

双向通道(Duplex-channel) 10.3.6

Switch 测试 9.4.2

数组下标(Array-index) 6.1.1

数组元素(Array-element) 6.1.1

T

通道(Channel) 10.3.2

条件循环结构(Conditional-loop Structure) 5.1.2

套接字(Socket) 11.1.4

跳转语句(Jump Statement) 5.2.1

U

Unicode(无国界编码方案) 2.2.6

UTF-8(Unicode 字符转换格式) 2.2.6

无限循环结构(Lnfinite-loop Structure) 5.1.3

W

外部包(Exterior Package) 8.5.1

X

线程(Thread) 10.2.1

协程(Coroutine) 10.2.1

相关(Association) 11.1.2

循环结构(Loop Structure) 5.1.1

形式参数(Formal Parameter) 7.3.1

消息机制(Message Communication Mechanism) 10.3.1

协议端口(Protocol Port) 11.1.2

选择结构(Selective Structure) 4.1.1

进程(Process) 10.2.1

Y

异步通道(Asynchronous-channel) 10.3.6

异常恢复机制(panic-and-recover) 7.6.3

用户数据报协议(User Datagram Protocol, UDP) 11.1.1

预留字段(Reserved-field) 8.1.1

运算符(Operator) 2.3.1

运行时管理(Runtime) 10.2.3

引用类型(Reference Type) 6.1.1

Z

字段(Field) 8.1.1

字段标签(Field-tag) 8.6.1

子集(Subset) 9.3.1

值类型(Value-type) 6.1.1

正式版(Official Version) 1.2.2

注释语句(Comment Statement) 3.2.3

整型(Integer) 2.2.2

作用域(Scope) 2.7.3

指针(Pointer) 2.2.7

M

枚举类型(Enumeration-type) 2.6.1

面向对象(Object Oriented, OO) 2.4.2

内存地址(Memory Address) 2.2.7

N

匿名函数(Anonymous-function) 7.5.1

匿名字段(Anonymous-field) 8.3.1

内置关键字(Built-in Keyword) 2.1.3

内置函数(Built-in Function) 7.2.4

O

OSI/RM(Open System Interconnection/Reference Model)

参考文献

- [1] Ivo Balbaert. The Way To Go[M]. Bloomington: iUniverse, 2009.
- [2] David Chisnall. The Go Programming Language[M]. Boston: Addison-Wesley Professional, 2012.
- [3] Mark Summerfield. Programming in Go[M]. Boston: Addison-Wesley Professional, 2012.
- [4] John P Baugh. Go Programming[M/OL]. Ann Arbor: The University of Michigan Press, 2010.
<http://www.goprogrammingbook.com/>
- [5] Jan Newmarch. Network Programming With Go[M/OL]. Cambridge: Aka E-books, 2012.
<http://jan.newmarch.name/go/>
- [6] Miek Gieben. Learning Go[M/OL]. Cambridge: Aka E-books, 2012.
<http://www.miek.nl/projects/learninggo/>
- [7] Russ Cox. Go Packages[DB/OL]. Mountain View: Google Inc., 2009.
<http://go-lang.cat-v.org/pure-go-lib/>
- [8] 许式伟. Go 语言编程[M]. 北京: 人民邮电出版社, 2012.
- [9] 谢孟军. Go Web 编程[M]. 北京: 电子工业出版社, 2013.
- [10] Douglas A Lyon. Java 程序员指南[M]. 朱剑平, 等, 译. 北京: 清华大学出版社, 2005.
- [11] Magnus Lie Hetland. Python 基础教程[M]. 北京: 人民邮电出版社, 2010.
- [12] 谭浩强. C 语言程序设计[M]. 北京: 清华大学出版社, 2005.
- [13] Stanford H. Rowe. 计算机网络(影印版)[M]. 北京: 清华大学出版社, 2006.
- [14] Silberschatz, Galvin, Gagne. 操作系统概念(影印版)[M]. 北京: 高等教育出版社, 2007.
- [15] 叶树华, 高志红. 网络编程实用教材[M]. 北京: 人民邮电出版社, 2006.
- [16] 任泰明. TCP/IP 网络编程[M]. 北京: 人民邮电出版社, 2009.



- ❖ 教学目标明确，注重理论与实践的结合
- ❖ 教学方法灵活，培养学生自主学习的能力
- ❖ 教学内容先进，反映了计算机学科的最新发展
- ❖ 教学模式完善，提供配套的教学资源解决方案
- ❖ 可下载教学资料：<http://www.tup.tsinghua.edu.cn>



清华大学出版社数字出版网站

WQBook  
www.wqbook.com

ISBN 978-7-302-34723-1



9 787302 347231 >

定价：39.00元